[redacted]

Math

Report No. UIUCDCS-R-72-550

AN APPROACH TO THE COMPILATION OF ARRAY EXPRESSIONS IN THE OL/2 LANGUAGE

by

Dale Rade Jurich

September 1972

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-72-550

AN APPROACH TO THE COMPILATION OF ARRAY EXPRESSIONS IN THE OL/2 LANGUAGE

by

Dale Rade Jurich

September 1972

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

[x] This work was supported in part by the National Science Foundation under Grant No. US NSF-GJ-328 and was submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science, September 1972.

ACKNOWLEDGMENT

I wish to sincerely thank the originator of OL/2, Professor J. R. Phillips, for his guidance, suggestions, and moral support which has lasted throughout my work on OL/2, and my entire academic endeavors in the field of Computer Science. I would also like to thank the past and present members of the OL/2 implementation team; Bob Bloemer, Barry Finkel, Cory Adams, John Latch, and John Gaffney for their cooperation and suggestions; and the University of Illinois at Urbana-Champaign for the support it has provided.

Furthermore, I am indebted to Mrs. Vivian Alsip, who did such a fine job of typing this report. Last, but not least, I wish to acknowledge my father, Rade Jurich, who first kindled my interest in the sciences.

TABLE OF CONTENTS

1. INTRODUCTION

OL/2, Operator Language version 2, has been designed as a natural language for array algorithms and has been implemented so as to insure efficient array operations for these algorithms. OL/2 provides a natural specification in that it is syntactically designed to adhere to the "language" of mathematics on which certain array algorithms are based. OL/2 has been implemented to provide for an efficient method of carrying out the operations defined in the language through the use of assembly language operational routines.

The philosophy behind the design and implementation of OL/2 involves handling the data arrays as units, rather than on an element by element basis as would be done in FORTRAN or PL/1. Hopefully, this philosophy will provide in practice an array processing system that will allow the convenient specification of array algorithms (minimizing parallel processes), while it will also provide for powerful array processing facilities which may very possibly suggest new breeds of array processing hardware organizations.

This thesis deals with the compilation of array expressions which form a most vital part of the OL/2 language. A more complete description of the language and its design philosophy as a whole can be found in the references [7, 8].

2. GENERAL STATEMENT PROCESSING

2.1 Statement Types

The OL/2 language contains various types of statements which reflect the array capabilities of the language as well as the usual program structure that is found in many procedure oriented languages. The following is a broad classification of the various types of statements.

1. Declarations or data definition statements.
2. Array partition and array set statements.
3. Array and scalar assignment statements.
4. Procedure definition and block structure control statements.
5. Program control statements (e.g., FOR, GO TO, and IF statements).
6. Input and output statements.

This thesis is primarily concerned with array assignment statements and array expressions which incidentally are used by the control statements and input-output statements. Appendix A contains a syntactic specification for OL/2 assignment statements and array expressions.

2.2 The Flow of Processing in the OL/2 Compiler

Figure 2.1 outlines the components of the OL/2 system as it is currently implemented on the IBM 360 at the University of Illinois. Basically, the TACOS compiler-compiler system (see Appendix A) is utilized to build a top down syntax directed compilation system. A linked list syntax table, built by the TACOS system for OL/2, is utilized by the TACOS parser/scanner module to control the syntax analysis of OL/2. At various points in the analysis, semantic action routines, written in PL/1, are called to undertake the semantic analysis and code generation for an OL/2 source program. The OL/2 compiler,

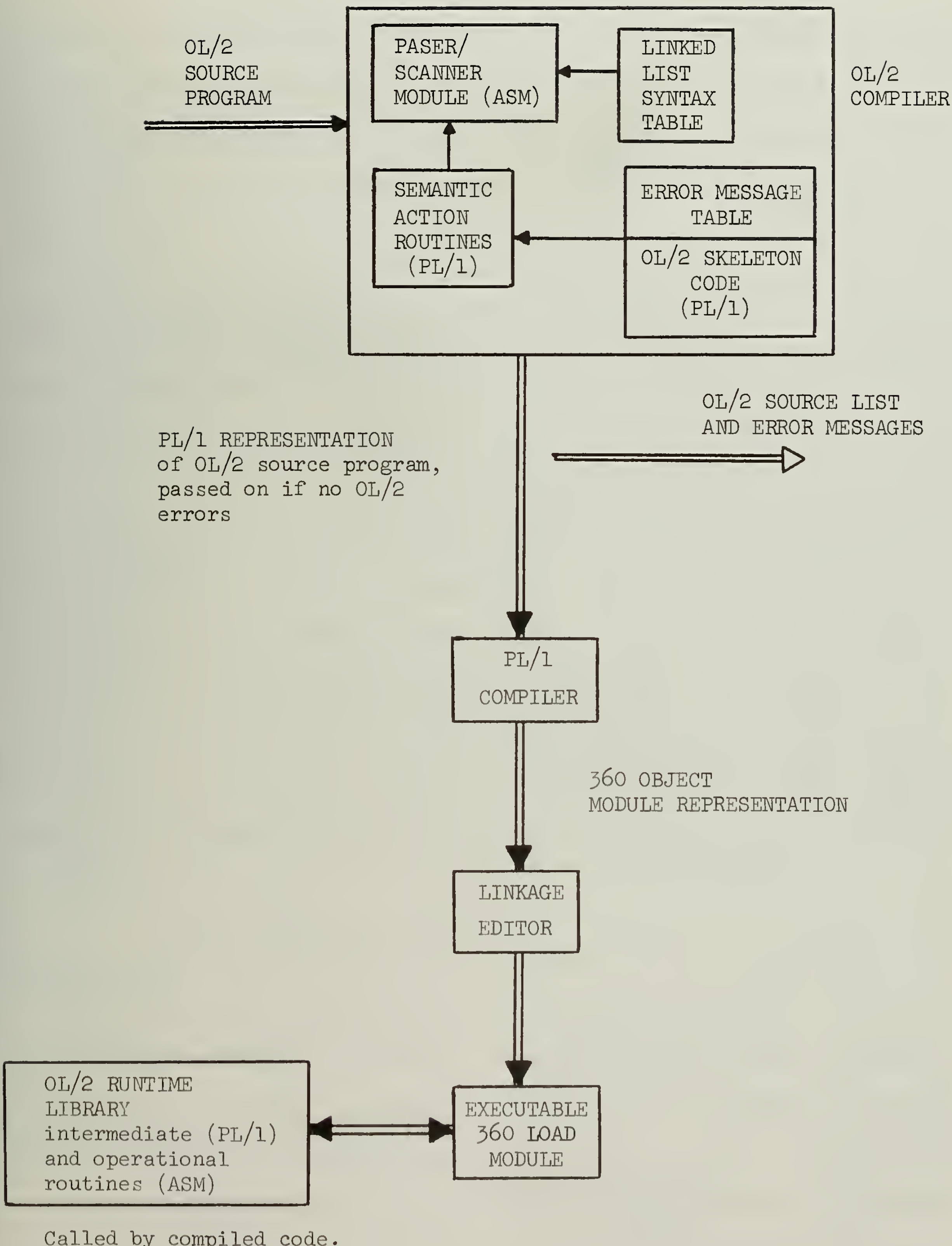


Figure 2.1. Block Structure of OL/2 System

which consists of the syntax table, the semantic action routines, the TACOS parser/scanner module, and various routines, compiles the OL/2 source code into an intermediate language, namely, PL/1. The purpose of the PL/1 is to act as a control language to set up appropriate information structures, to redefine the data structures for all arrays, and to link these structures to OL/2 run-time routines that control the storage allocation and all array operations. The run-time routines are written in assembly language and provide efficient processing for array operations. Many of the ordinary tasks are also delegated to the PL/1 compiler, such as, scalar expressions, function evaluation, and block structure control.

Figure 2.2 outlines the flow of processing in the OL/2 semantic action routines. Statement type determination is done as early in the syntax analysis as possible, and an appropriate module controls the compilation of a given statement type. The compiler is not completely moduler in that interaction between sections of the semantic action routines does exist, and many routines common to several modules are utilized. The section of the compiler, or module, that is described here is the semantic action module that compiles OL/2 array expressions.

2.3 Array Expression Compilation

In OL/2, array expressions can be imbedded in various types of statements, such as FOR statements, IF statements, assignment statements, procedure statements, and input-output statements. Whenever an array expression exists within an OL/2 statement, the expression processing module is activated to produce (PL/1) "intermediate" object code for the expression.

As Figure 2.2 indicates, array expression processing can be divided into three phases within the expression module. The first phase involves the

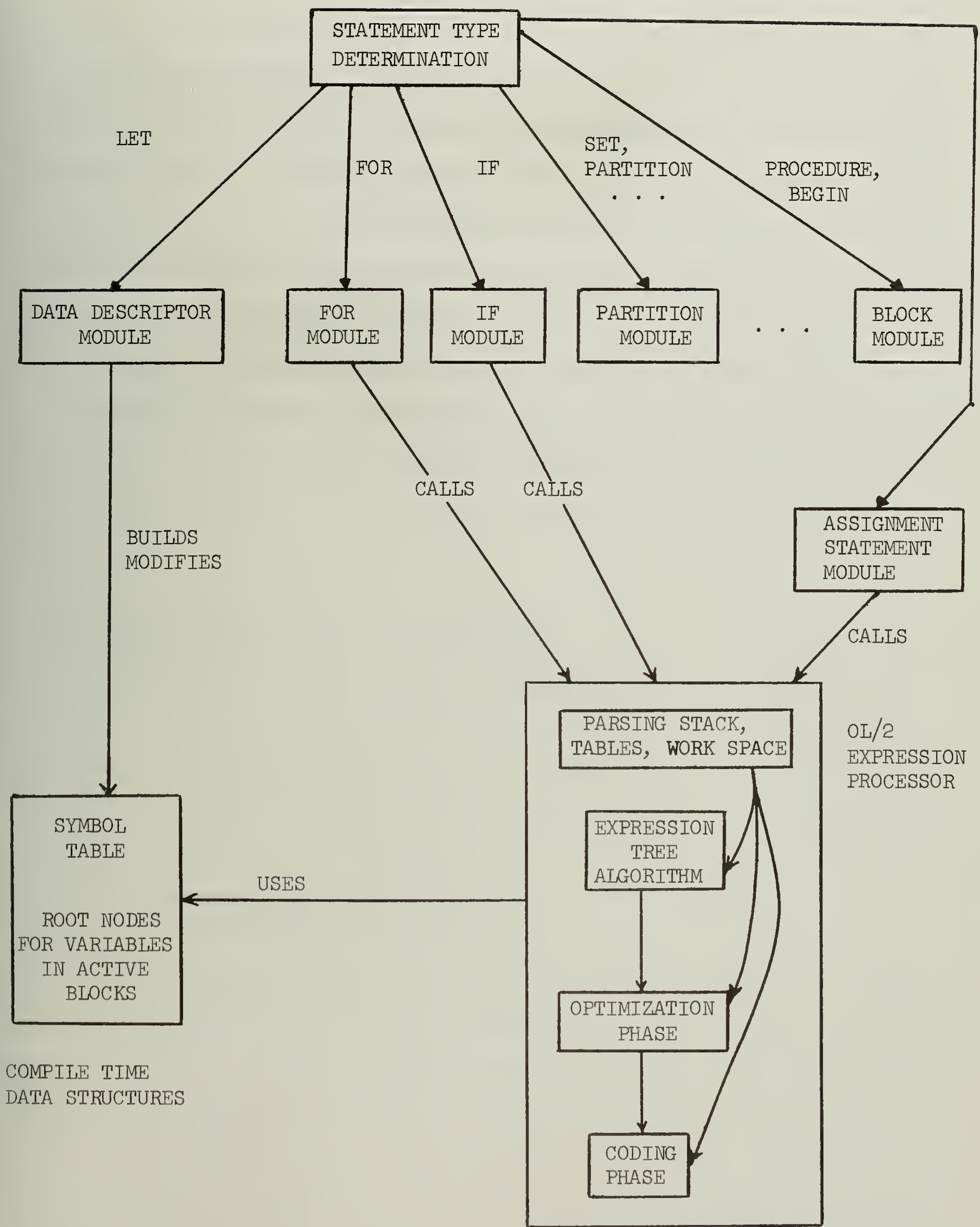


Figure 2.2. Semantic Action Processing of OL/2 Statements

syntactic and semantic analysis of an expression, and converts the source code representation into an expression tree intermediate code representation. The second phase modifies the expression tree so as to allow an optimized coding of the expression by the third phase, the coding routine. The coding routine transforms the optimized intermediate expression tree representation into a sequential list of three operand code (operand, operand, result) that is the PL/1 intermediate object code for the expression. Sections 3, 4, and 5, describe the algorithmic structure of the three expression processing phases of the OL/2 compiler. Another coding algorithm that utilizes a slightly different approach to the handling of array expressions in OL/2 is described in LATCH [5].

3. EXPRESSION PARSING

3.1 Goals of the Expression Parser

As was previously outlined, OL/2 array expressions are imbedded in many of the common OL/2 statements. During compilation, a set of modules is called at each occurrence of an OL/2 expression to translate the source expression into "intermediate code", (the format of the code produced is described in section 5). This section will discuss the process of translating the source OL/2 expressions into an expression tree form that is utilized by the optimizating and coding phases of the compiler. During the tree building phase, syntactic error checking is done, as well as some semantic error checking, and optimization in the form of repeated subexpression handling is also done.

3.2 Basic Mathematical Objects and Operations

Figure 3.1 outlines the major operand types and operations that make up OL/2 expressions. The classes of objects or variable types that can be involved in an OL/2 expression are scalars, vectors, and operators. The first two classes of objects are defined in the standard sense, while the third term is used to denote the arrays of various orders or dimensionalities that are used to represent the corresponding mathematical operator. For example, a linear operator used in the above context means a matrix. When the term array is used, the objects involved may be of type vector or matrix, since in OL/2 vectors are treated, from the implementation point of view, like "degenerate" two dimensional arrays. In the figures, and elsewhere in this paper, the following conventions are used: α, β, γ denote scalars; x, y, z denote vectors; and A, B, C denote operators (matrices and higher dimensional arrays). In some cases the results in the figure apply to operators of dimensionality

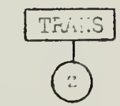
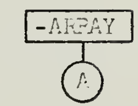
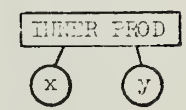
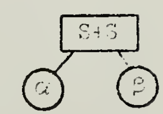
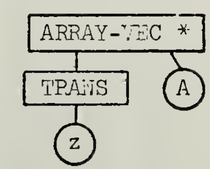
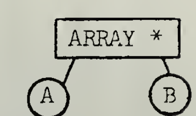
PRIORITY LEVEL	OL/2 SYMBOL	ABSTRACT SYMBOL AND MEANING	OPERAND CLASS 1	OPERAND CLASS 2	RESULT CLASS	SOURCE EXAMPLE	ABSTRACT EXAMPLE
2	'	TRANS; vector or array transpose	MATRIX COL VECTOR ROW VECTOR	--- --- ---	MATRIX ROW VECTOR COL VECTOR	z'	
2	+,-	+ARRAY,-ARRAY; unary negation of each array element (ignore +)	ARRAY	---	ARRAY	-A	
3	(·,·) or *	INNER PROD; vector inner product	ROW VEC	COL VEC	SCALAR	(x,y) or x'*y	
3	||·||	NORM; various vector and matrix norms	VECTOR MATRIX	--- ---	SCALAR SCALAR	||x||	
4	*,+,-,**,/	S*S,S+S,S-S,S**S,S/S; normal (i.e., PL/1 defined) scalar operations	SCALAR	SCALAR	SCALAR	$\alpha+\beta$	
5	*	ARRAY-VEC *; array times column vector or row vector times array	MATRIX ROW VEC	COL VEC MATRIX	COL VEC ROW VEC	z'*A	
6	*	ARRAY*; normal array multiplication and the vector outer product	MATRIX COL VEC	MATRIX ROW VEC	MATRIX MATRIX	A*B	

Figure 3.1. Basic OL/2 Operations

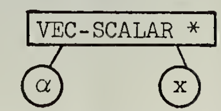
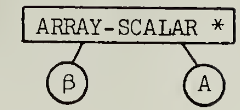
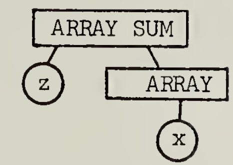
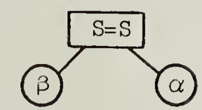
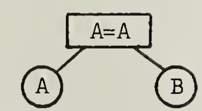
PRIORITY LEVEL	OL/2 SYMBOL	ABSTRACT SYMBOL AND MEANING	OPERAND CLASS 1	OPERAND CLASS 2	RESULT CLASS	SOURCE EXAMPLE	ABSTRACT EXAMPLE
7	*	VEC-SCALAR *; vector times scalar	VEC SCALAR	SCALAR VEC	VEC VEC	α*x	
8	*	ARRAY-SCALAR *; array times scalar	MATRIX SCALAR	SCALAR MATRIX	MATRIX MATRIX	A*B	
10	+,-	ARRAY SUM; array ± array	ARRAY	ARRAY	ARRAY	z-x	
11	=	S=S; assignment scalar ← scalar	SCALAR	SCALAR	SCALAR	β=α	
11	=	A=A; assignment array ← array	ARRAY	ARRAY	ARRAY	A=B	

Figure 3.1 (continued). Basic OL/2 Operations

greater than two, although these objects are not specifically mentioned, since the compilation process is essentially the same.

The priority levels given in Figure 3.1 define the order in which the evaluation of an expression takes place, with the lowest priority operations being evaluated first. The priorities were chosen in order to minimize the number of operations when evaluating array expressions, and the tree structure built up by the parser reflects this ordering. For example, in an expression of the form

$$A * B * \alpha * \beta * y$$

the priorities assigned are equivalent to the forced priority evaluation:

$$(A * (B * ((\alpha * \beta) * y))).$$

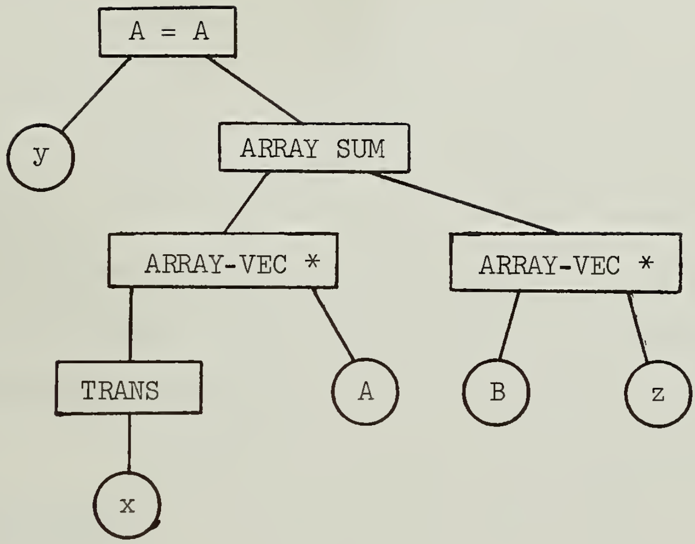
By recursively applying the "rules" given in Figure 3.1, while also allowing forced priorities with the use of parentheses, all basic OL/2 expressions can be generated, and thus the rules serve as a simplified abstract syntax for an OL/2 expression. By combining the abstract representations for each operation, keeping in mind the evaluation priorities, the abstract tree representation for the expression can be formed. The basic task of the expression parser is to build a concrete representation of the tree using PL/1 based variable structures as nodes and pointer variables as links. This concrete representation is then used by subsequent compiler phases. Syntactic and semantic error checking is also done during the parsing phase so that subsequent phases can assume an essentially error free representation. Examples of OL/2 source expressions and the corresponding abstract expression tree representations are given in Figure 3.2.

3.3 Concrete Representation of Nodes

PL/1 structures that represent all known information concerning each

EXPRESSION: $y = x' * A + B * z$

ABSTRACT
EXPRESSION TREE:



EXPRESSION: $|| A * x || * B * C$

ABSTRACT
EXPRESSION TREE:

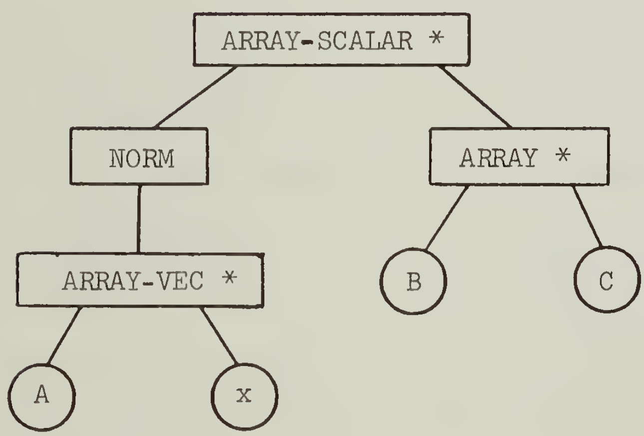


Figure 3.2. OL/2 Abstract Trees

distinct OL/2 variable defined in a program are built up by a definition processing module group. These structures are available to the expression module when an expression is to be parsed, in complete form since a "definition before use" one source pass system is used. During the definition processing phase, a symbol table is built up that contains the variable names of all variables active in the current OL/2 block being scanned. New entries are added at the bottom of the symbol table, and thus the table is searched bottom up to locate the proper variable associated with a specific OL/2 name. When the "end" statement of an active block is encountered, the symbol table is cleared of all definition information built up in the block, since in a one pass environment it is no longer needed.

The PL/1 representation of an operand node, and a definition of the node fields are given in Figure 3.3 and Table 3.3. Operand nodes are linked together to form the expression tree by operator nodes that define the specific operation involved and contain information fields that are utilized by the parser, optimizer and coder phases. Figure 3.3 and Table 3.3 also present the PL/1 representation of an operator node and give a definition of the information contained in the node fields.

It should be noted that operand nodes can be "reused", possibly within a single expression, since the information that they contain is static, in the sense that it does not change within the block that the OL/2 variable is active. Operator nodes contain information that is dependent on the context in which the operator appears and therefore must be created for each occurrence of an OL/2 operational symbol.

3.4 Expression Tree Building Algorithm

In the current OL/2 system, syntax analysis proceeds in a top down

NODE_PTR →

LLINK	RLINK
SEQ_#_PTR	STRING_PTR
$#DIMENSIONS	PART_SIZE
#_OF_TIMES_TO_USE	TYPE_EXP
$TYPE_CODE	NEGATE_TAG
TRANSPOSE_TAG	IDENTITY_TAG

```
1   TREE_NODE BASED (NODE_PTR),
    (

        2   RLINK,
        2   LLINK,
        2   SEQUENCE_#_PTR,
        2   STRING_PTR                 )   POINTER,

    (

        2   $#DIMENSIONS,
        2   PART_SIZE,
        2   #_OF_TIMES_TO_USE,
        2   TYPE_EXP,
        2   $TYPECODE                  )    FIXED BINARY
                                              (15.0),

    (

        2   NEGATE_TAG,
        2   TRANSPOSE_TAG,
        2   IDENTITY TAG               )    BIT (1)
```

Figure 3.3. Concrete Node Structure

DEFINITIONS:

FIELD NAME	OPERAND NODE	OPERATOR NODE
RLINK	Null pointer	Pointer to right subtree
LLINK	Null pointer	Pointer to left subtree
SEQ_#_PTR	Pointer to expression string defining an array sequence element	Null pointer
STRING_PTR	Pointer to string containing name of operand	Null pointer
$#DIMENSIONS	Number of dimensions of operand: 0 Scalar 1 Vector 2 or more Operator	Number of dimensions of result of expression evaluation below node
PART_SIZE	Used by partitioning routines	
#_OF_TIMES_TO_USE	Number of times a subexpression is repeated	
TYPE_EXP	Type of operand or expression 0 Scalar 1 Function 2 Column Vector 3 Row Vector 4 Matrix 5 Null Operand 6 Boolean Result	
$TYPE_CODE	Not defined	Type code of operator
NEGATE_TAG	Operand is to be negated	Not defined
TRANSPOSE_TAG	Operand is to be transposed	Not defined
IDENTITY_TAG	Operand is an identity operator (do not allow an assignment to it)	Not defined

Table 3.3. Node Field Descriptions

manner, with various semantic action routines being called at specific points in the analysis. Appendix A contains a short description of the TACOS compiler-compiler system, and the OL/2 compiler that was developed using the TACOS system. For the purposes of this section it will be assumed that expression analysis proceeds in a left to right manner, with the required semantic routines called at the appropriate times. In this way, the actual mechanisms of the top down analysis scheme become transparent to the description of the tree building process.

Two structures are important to the tree building process; the node stack, and the precedence matrix. The node stack in the abstract sense consists of two cells per level, as shown in Figure 3.4. The cell labeled SUBTREE_PTR is a PL/1 pointer variable that points to the root node of the subtree, which may be a single node, described by a given cell in the stack. The variable SUBTREE_TYPE associated with a given cell holds a number that represents the type of object represented by the subtree described by the stack cell. The legal types are: scalar, function, column vector, row vector, matrix, null operand and boolean result. Figure 3.4 presents an example of a stack cell that represents a vector subpart of an expression. The tree building algorithm described in this section is concerned, therefore, with stacking operands in a left to right expression analysis, building subtrees from stacked operands when possible, and utilizing the precedence rules given earlier. Eventually this process will result in a single stack entry that describes the minimum operation evaluation tree for the entire expression being considered.

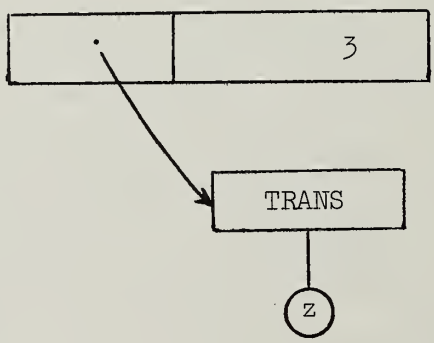
Since the multiply symbol "*" is used in the source language to represent several different priority operations, i.e., scalar * scalar, column vector * matrix, matrix * matrix, matrix * row vector, column vector * row vector, and row vector * column vector, some type of semantic test on the

SUBTREE_PTR	SUBTREE_TYPE
Pointer to root of subtree	Type of expression that results from evaluation of a subtree

TYPE NUMBERS:

0 SCALAR
1 FUNCTION
2 COLUMN VECTOR
3 ROW VECTOR
4 MATRIX
5 NULL OPERAND
6 BOOLEAN RESULT

(a). Expression Stack Cell



(b). Expression Stack Cell for a Column Vector Expression

	SCALAR	FUNCTION	COL VEC	ROW VEC	MATRIX
SCALAR	0	1	1	1	1
FUNCTION	1	1	1	1	1
COL VEC	1	1	2	1	2
ROW VEC	1	1	0	1	0
MATRIX	1	1	0	1	1

0 = DON'T UNSTACK
1 = UNSTACK
2 = ILLEGAL

(c). Precedence Matrix

Figure 3.4. Expression Algorithm Data Structures

operand types is necessary during compilation to determine the actual priority of a given * encountered in a left to right analysis. Since the operand types are available in SUBTREE-TYPE entries in the stack cells, it is natural to assign a relative precedence to the "*" operation, which is to be applied to the top two stack operands, when an unstack or combining operation is possible as far as the syntax analyzer is concerned. For this purpose, a precedence matrix is defined for the "*" operation in Figure 3.4. This is based on the types of the top two stack operands and indicates when an unstack may or may not be done, based on the operands already seen on the left, and those that may appear on the right of a given point in the expression parse. The use of the table will become clearer when the tree building algorithm and examples of its usage are presented.

The expression tree building algorithm is a basic stack precedence scheme, and is similar to the scalar expression parsing algorithm described in Hopgood [3]. The addition of the multiply precedence matrix, and error checking are the main differences. It should be noted that as operands are unstacked and subtrees of the complete expression tree are created, it is necessary to define temporary variables which are needed to complete the expression evaluation at run time. For example, for the subtree given in Figure 3.5, the operator node

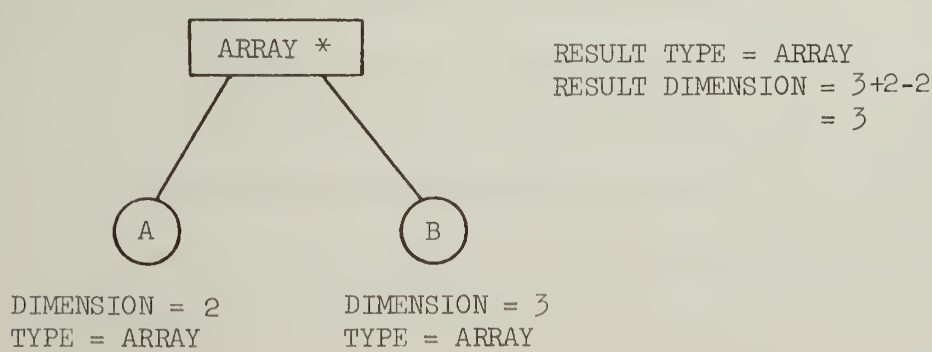


Figure 3.5. Temporary Results

represents the result of the operation, i.e., in this case a temporary matrix. During parsing, sufficient information is available to fill in the number of dimensions (but not its size) and the type code node fields of the operator/result node in all cases. This information is used by later phases of the compiler, and will be discussed in subsequent sections. The expression tree algorithm is as follows:

ET1: Extract the first operand and stack it.

ET2: Follow top down syntax analysis based on the next operator symbol. If illegal construct, set error flag and try to continue. If done with expression exit to next phase.

ET3: Extract the next operand and stack it.

ET4: Can an unstack on the current operator symbol class be tried as far as the syntax analyzer is concerned? If yes, go to ET5; otherwise, go to ET2.

ET5: If the current operator is in the class multiply, go to ET6; otherwise, go to ET7.

ET6: Consult multiply unstacking precedence table using operand type classes. If "Unstack" = 1 entry in table, go to ET7; otherwise, go to ET2.

ET7: Go to RS2 of the repeated subexpression algorithm, described in section 3.5. If semantic error due to dimensionality or type of operands, set error flag and try to continue. Build an operator node, link top operand node(s) to operator node, delete top operand node(s) from stack, and replace with operator node that represents a subtree of the completed tree. Fill in dimensionality of result node. Go to ET4.

3.5 Example Parse with Semantic Error Checking

The steps involved in parsing an OL/2 expression will be described to illustrate the use of the tree building algorithm with error checking. Consider the array expression

$$x' = \alpha * y + y' * (A + B);$$

where x and y are column vectors, A and B matrices, and α is a scalar. The following events occur:

1. The operand x is found and identified as a column vector, through a symbol table look-up, and placed in the stack (see Figure 3.6(a)).
2. The transpose operation is recognized as applying to x, an operator node is built, and the new subtree replaces the former one in the stack (see Figure 3.6(b)).
3. The equal sign is recognized and the syntax analyzer recurses on it, assuming the statement found is an assignment statement, and goes to look for a vector expression on the right side.
4. α is identified as a scalar, and a node is stacked (see Figure 3.6(c)).
5. The syntax analyzer picks off the '*' and recurses to look for an OL/2 factor. No unstacking can be done at this point.
6. The operand y is found and identified as a column vector. A stack entry is pushed for y (see Figure 3.6(d)).
7. '+' is found which completes the OL/2 factor, and unstacking is done (after checking the precedence table) on the scalar and matrix. An operator node is created and pushed for the result (see Figure 3.6(e)).

8. Since a '+' has been recognized, an OL/2 term is sought by the syntax analyzer.
9. y is picked off and stacked as a column vector operand (see Figure 3.6(f)).
10. The transpose operator is recognized as applying to y, and a new row vector type subtree node is stacked (see Figure 3.6(g)).
11. '*' is picked off and the syntax analyzer looks for an OL/2 factor.
12. The syntax analyzer recurses one level for the '(', looking for a parenthesized construct.
13. A is recognized as an OL/2 matrix variable and a node is stacked (see Figure 3.6(h)).
14. '+' is found, an OL/2 term is sought.
15. B is recognized as a matrix variable and a node is stacked (see Figure 3.6(i)).
16. The closing ')' is found completing the OL/2 term. Unstacking is done, an operator node of type ARRAY SUM is built and replaces the nodes for A and B in the stack. Error checking is done to assure that the dimensions of A and B are compatible. The dimension of the result is inserted into the operator node (see Figure 3.6(j)).
17. The syntax analyzer recognizes ';' as a statement delimiter and begins to return up its parse tree.
18. y' * (A+B), row vector times matrix, constitutes a legal OL/2 factor, and an unstack is done, with an operator node representing a row vector being pushed onto the expression stack (see Figure 3.6(k)).

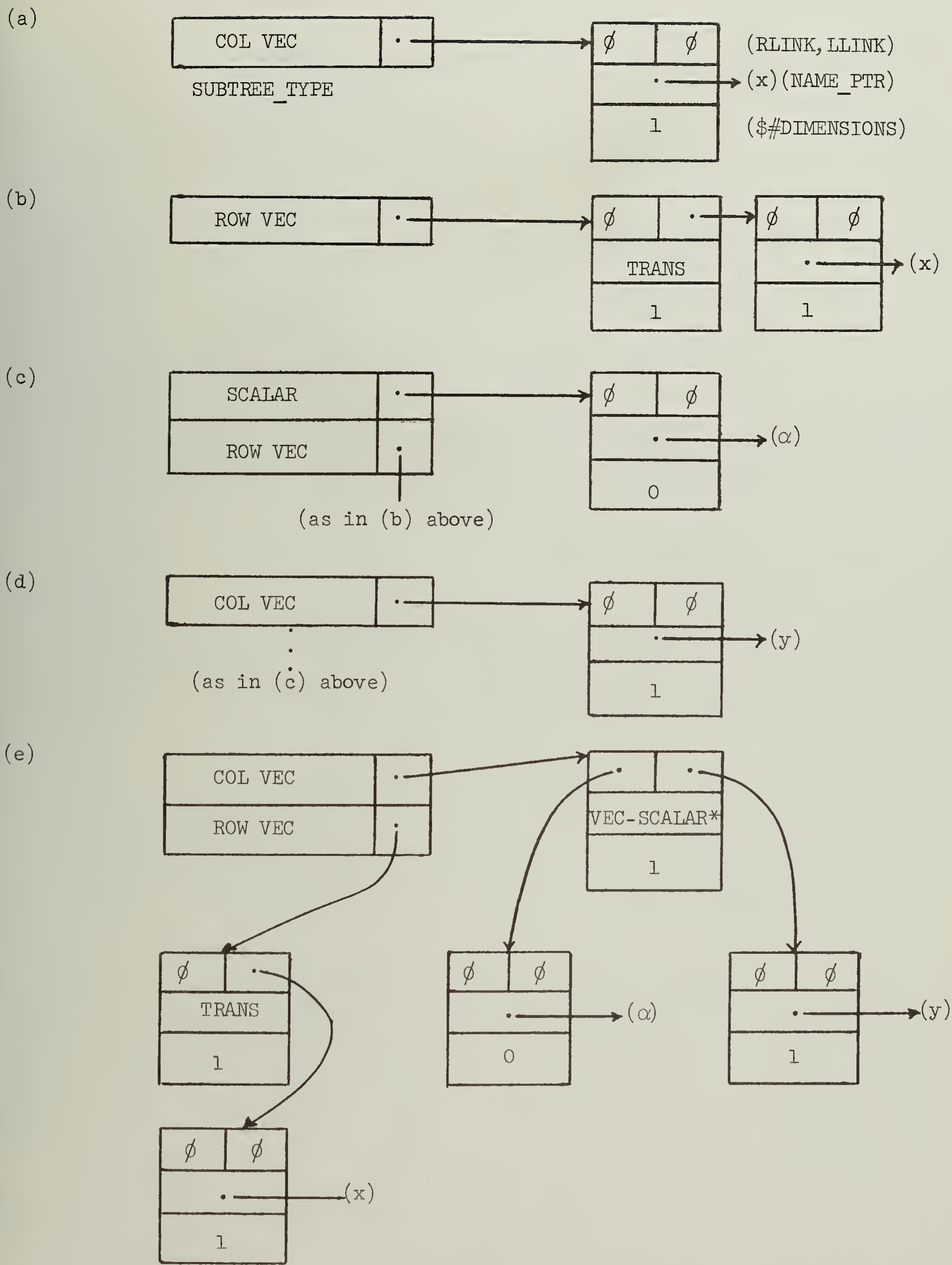


Figure 3.6. Stack and Node Contents for Example Parse

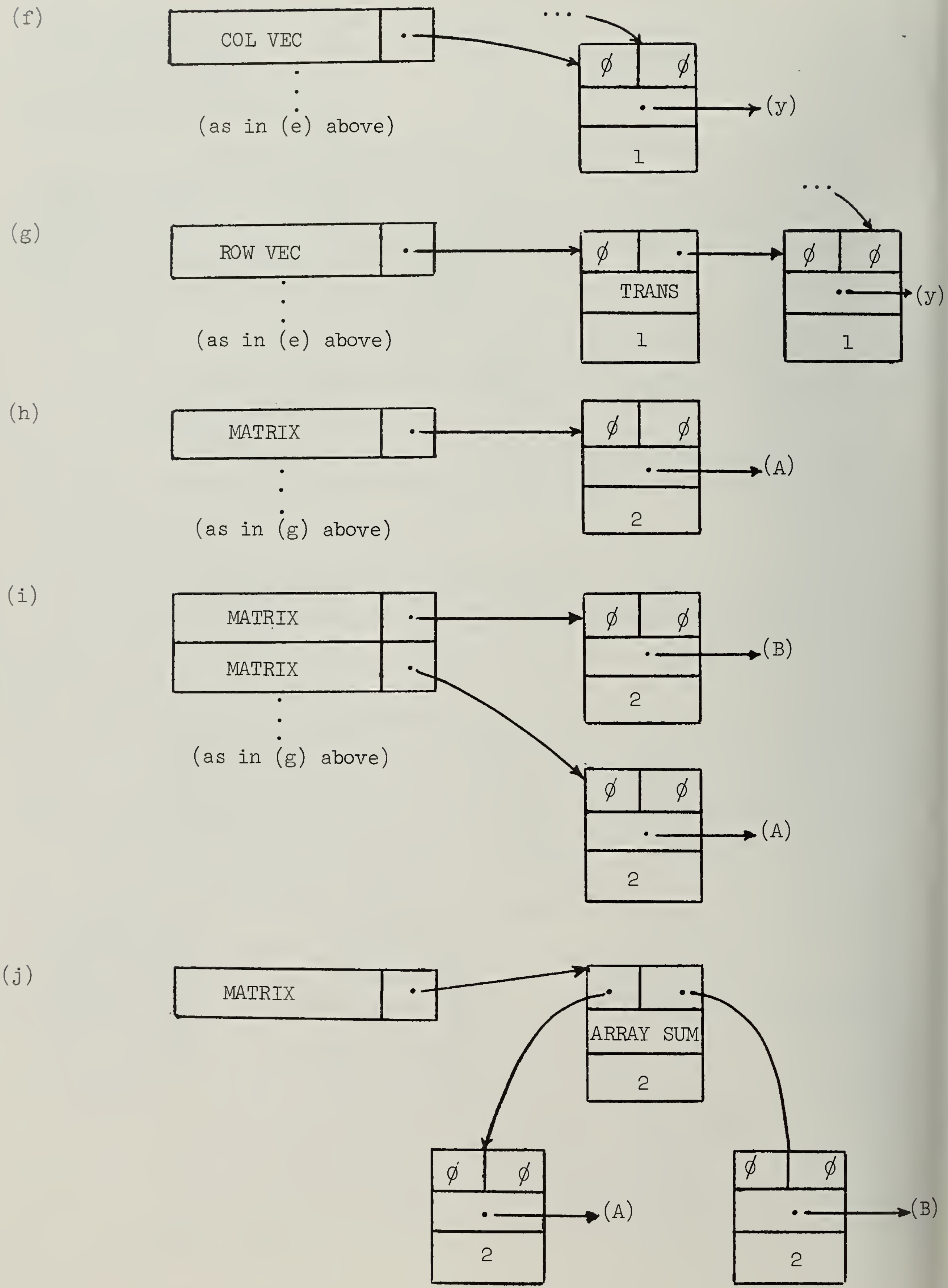


Figure 3.6 (continued). Stack and Node Contents for Example Parse

(k)

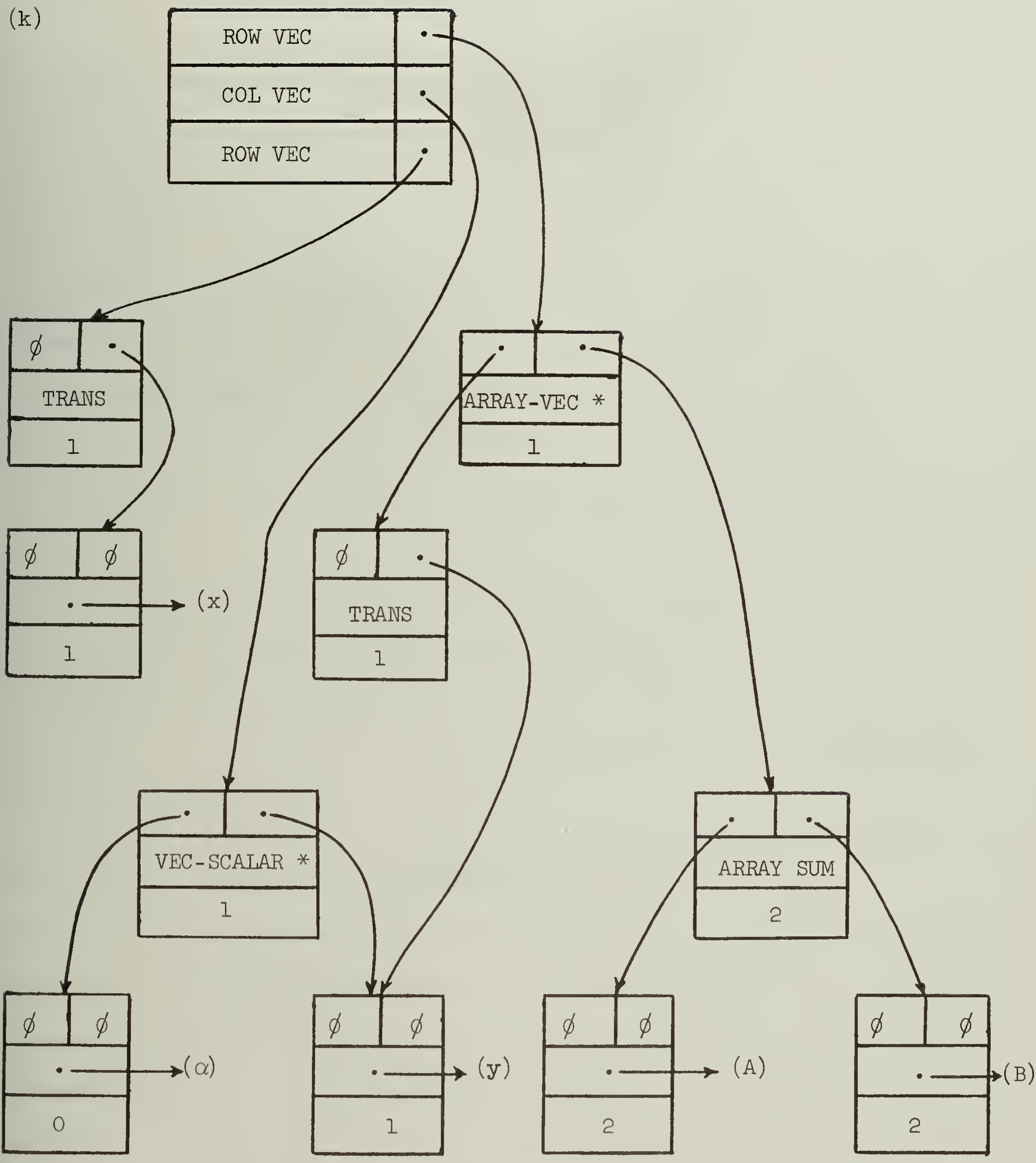


Figure 3.6 (continued). Stack and Node Contents for Example Parse

19. The syntax analyzer now tries to complete an OL/2 term involving a vector addition.

20. The two top stack entries are found to be of types row vector and column vector, respectively, when an unstack is tried. The semantic error checking routines signal an error, since a row vector added to a column vector constitutes an illegal OL/2 subexpression. Further processing of the statement is blocked by the setting of error flags, e.g., no coding should be attempted, and appropriate error messages are written into an error file with information concerning the statement number, type of error encountered, and the probable location within the statement of the error.

Normally the syntax analyzer tries to completely analyze an expression, so that all errors may be uncovered, but the expression tree is considered non-codeable after a single error although the building processes are continued so that further semantic error checking may be done.

3.6 Repeated Subexpression Handling

A repeated subexpression is defined in the context of OL/2 as follows: a repeated subexpression, within an OL/2 expression, exists when two subtrees of an expression tree are operationally identical. An example of an OL/2 repeated subexpression is illustrated in Figure 3.7. The expression in Figure 3.8 does not contain a repeated subexpression, although subparts of the expression are repeated, due to the "minimum operation" type tree that is formed. Repeated subexpressions are an important factor to consider in OL/2 expressions because of the number of operations that may be involved in evaluating array expressions. If a repeated subexpression is not recognized and processed in some way, code

α*A*B+C*A*B

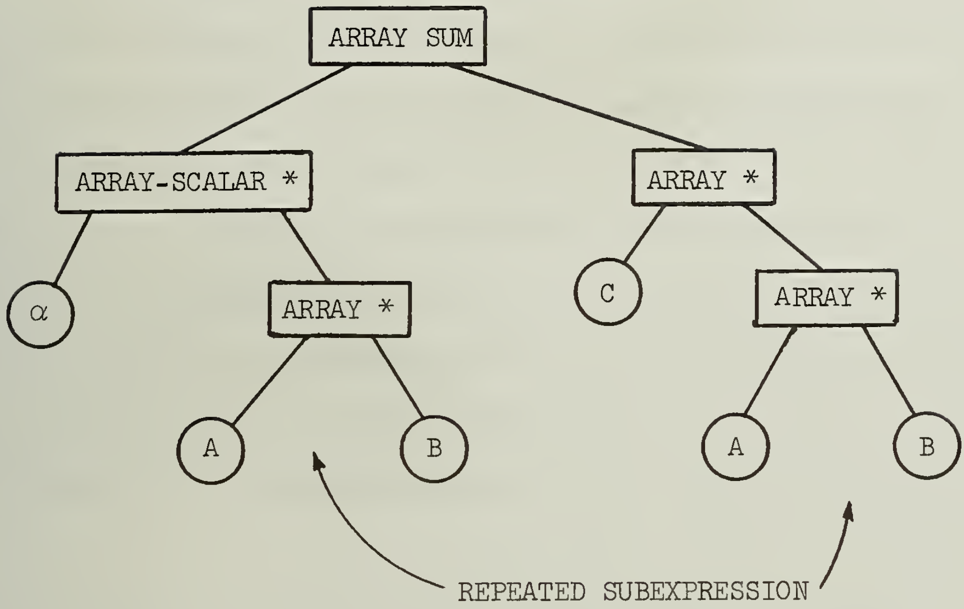


Figure 3.7. Repeated Subexpression Example

A+B*x+A*B

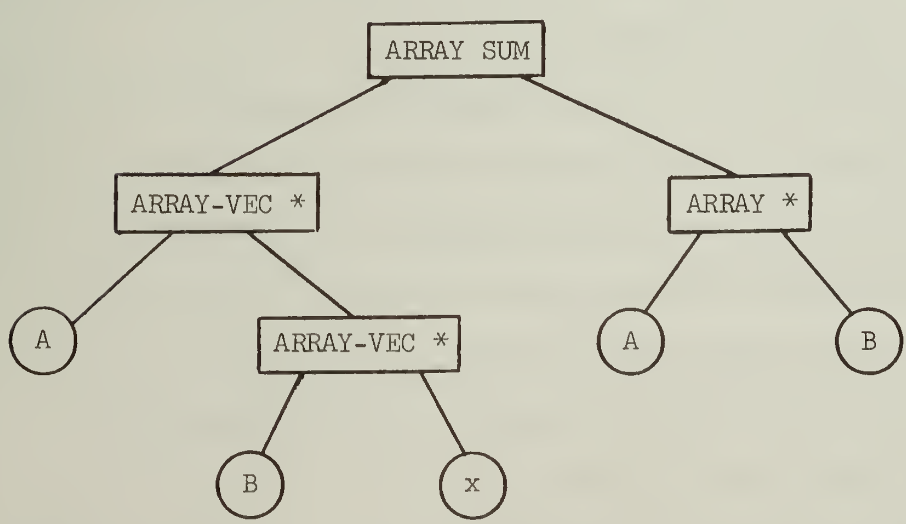


Figure 3.8. No Repeated Subexpression

will be generated for the evaluation of the expression represented by each of the duplicate subtrees and this is unacceptable in an array language.

The approach in OL/2 to processing repeated subexpressions involves saving the temporary result due to the evaluation of one appearance of the subexpression, and using this temporary result later in the expression evaluation. A storage and computation time tradeoff must be considered in this approach since the temporary variable representing the subexpression must be maintained in storage at run time in order to avoid recomputing the temporary result. Since processing time is the major concern when dealing with large arrays, assuming a storage scheme involving secondary devices is available, the implementation just described seems justified.

The tasks of the expression parser in repeated subexpression handling are outlined below.

1. Recognize the existence of repeated subexpressions.
2. Restructure the expression tree so that repeated subexpressions are evaluated only once.
3. Mark the repeated subexpression when it is no longer needed (used in the coding phase).

The first and third tasks are accomplished by an algorithm based on that of Hopgood [3]. Figure 3.9 describes the data structures involved in the algorithm. The node field #_OF_TIMES_TO_USE in an operation node contains the number of appearances of the expression represented by the subtree for which the operation node is the root. This node field is filled in by algorithm RS below, and provides the information needed for third task above to be satisfied. The repeated subexpression algorithm is as follows:

RS1: Set the CURRENT-TABLE-POINTER to zero before the processing of each OL/2 expression.

RS2: (Entered from the tree building algorithm at ET7 when an unstack operation is called). Is the current operation associated with the unstack, subtree building, step allowed in an OL/2 repeated subexpression? If yes, go to RS3; otherwise, go to RS6.

RS3: Search the SUBTREE-TABLE entries, from the first to the current entry, until the following conditions are met:

1. POINTER_TO_OPERAND_1 = pointer value on top of expression stack,
2. POINTER_TO_OPERAND_2 = pointer value on second level of expression stack,
3. OPERATION = current operation involved in the subtree about to be built, i.e., involved in unstack operation of expression tree algorithm.

If a match is found, go to RS4; otherwise, go to RS6.

RS4: In the result node, increase the field value #_OF_TIMES_TO_USE by one.

RS5: If RLINK = ∅ for either or both the first and second stack entries, then subtract 1 from #_OF_TIMES_TO_USE for either or both respective nodes referenced by the stack entries--A multilevel repeated subexpression has been found. Delete the top two stack levels and replace them by POINTER_TO_RESULT from the matching SUBTREE_TABLE entry found. Go to ET4 (tree building algorithm) which skips the normal unstack step.

RS6: Increase value of CURRENT_TABLE_POINTER by one.

RS7: Add a new SUBTREE_TABLE entry as follows:

1. POINTER_TO_OPERAND_1 = pointer value on top level of expression stack,
2. POINTER_TO_OPERAND_2 = pointer value on second level of expression stack,
3. POINTER_TO_RESULT = pointer value of the operation node associated with the subtree being built,
4. OPERATION = value of representative symbol for the current operation being considered.

RS8: Carry out ET7 as outlined in the expression tree algorithm. The algorithm above is slightly modified for the case of a unary operation, with the top level of the expression stack considered rather than the top two levels as in the case of a binary operation.

The abstract tree representation used in the case of a repeated subexpression is best illustrated by the example in Figure 3.10. The repeated expression appears once in the tree, and is linked once for each occurrence that appears in a non-repeated representation. In Figure 3.10, the operator node representing the result of A * B has #_OF_TIMES_TO_USE set to 2.

Certain operations such as PART_OF or SEQUENCE (described in section 3.7) cannot be considered as appearing in repeated subexpressions since these modifiers can define different operands at run time even though they modify the same OL/2 variable at compile time. Also, operations involving scalar variables only are not considered since the extra storage requirements for the temporary variables generated are negligible.

3.7 Other OL/2 Operations Available

Many OL/2 operations other than the basic operations described in

POINTER_TO_OPERAND_1
POINTER_TO_OPERAND_2
POINTER_TO_RESULT
OPERATION

SUBTREE_TABLE entry

```
1  SUBTREE_TABLE (50),
   (
        2    POINTER_TO_OP1,
        2    POINTER_TO_OP2,
        2    POINTER_TO_RESULT
                              )  POINTER,
        2    OPERATION FIXED BINARY (15,0)
```

CURRENT_TABLE_PTR - Array index used to reference latest SUBTREE_TABLE entry

Figure 3.9. Data Structure for Repeated Subexpression Algorithm

α*A*B+C*A*B

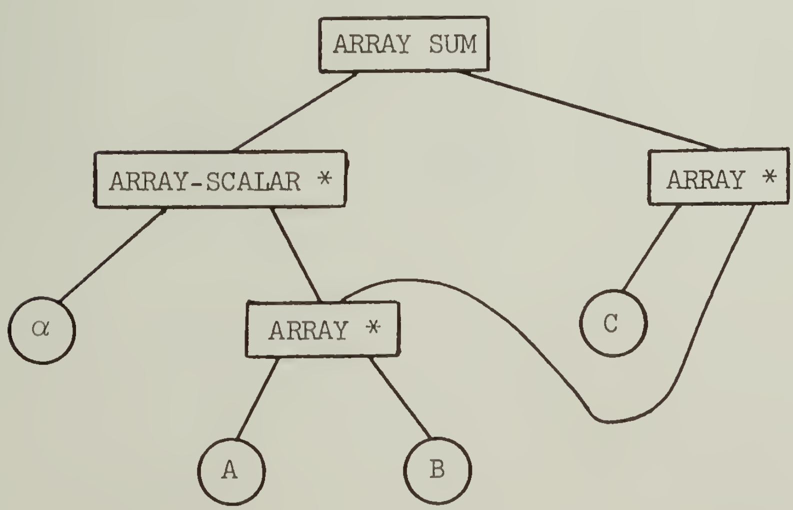


Figure 3.10. Abstract Tree Representation of a Repeated Subexpression

section 3.2 are available. From the compilation point of view, these nonbasic OL/2 operations are handled in the same manner as the basic operations. That is, operator nodes are built up to represent the operator and expression subtrees are created to represent the operator/operand combinations. Figure 3.11 describes some of the nonbasic OL/2 operations that have been implemented and illustrates the corresponding abstract tree structures for the operations. Reference [7] describes the complete set of OL/2 operations currently implemented and explains their uses. It can be noted that any unary or binary operation (and some n-ary operations $n > 2$) fit well into the expression tree format used in the OL/2 compiler, which eases the task of implementing compilation facilities for new OL/2 operations greatly.

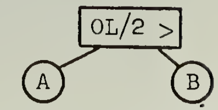
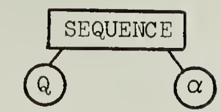
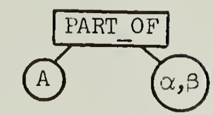
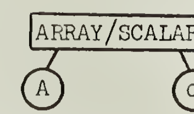
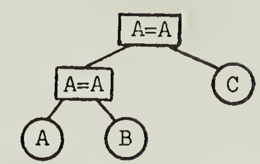
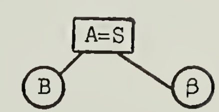
PRIORITY LEVEL	.OL/2 SYMBOL	ABSTRACT SYMBOL AND MEANING	OPERAND CLASS 1	OPERAND CLASS 2	RESULT CLASS	SOURCE EXAMPLE	ABSTRACT EXAMPLE
	>, <, =, ¬=, ¬>, ¬<	OL/2 >, OL/2 <, ...; compare arrays element wise with each other, or with a single scalar	ARRAY OR SCALAR	SCALAR OR ARRAY	BOOLEAN VALUE	A > B	
1	[·]	SEQUENCE; denotes an element of a sequence of arrays	ARRAY SEQUENCE VARIABLE	SCALAR	ARRAY	Q[α]	
1	< · >	PART_OF;	ARRAY	SCALAR LIST	ARRAY	A < α, β >	
1	(·)	ELEMENT_OF; denote a row column expression of an array element	ARRAY	SCALAR LIST	SCALAR	y(α)	
9	/	ARRAY/SCALAR; array * (1/scalar)	ARRAY	SCALAR	ARRAY	A/α	
11	,	A=A; multiple left hand side of array assignment	ARRAY	ARRAY	ARRAY	A, B=C	
11	=	A=S; assignment, each element of an ← scalar value	ARRAY	SCALAR	SCALAR	B=β	

Figure 3.11. Nonbasic OL/2 Operations

4. TEMPORARY STORAGE MINIMIZATION

In an array language that is designed to operate on large arrays, there is the important problem of minimizing temporary storage when evaluating array expressions. Even if a large amount of secondary storage is available for temporary arrays, the number of input-output requests may increase prohibitively if care is not used in reducing the number of temporary arrays. This chapter will be concerned with some techniques for reducing temporary array variables and with pointing out some unusual problems which arise in an array language such as OL/2.

The problem of compiling array expressions so that a minimum amount of temporary storage is utilized at run time for expression evaluation is a difficult one to solve. The difficulties arise from two sources.

1. The complete structure of an array expression must be known for minimization to be attempted, and this structure is not known to the compiler during initial expression parsing.
2. In OL/2 with dynamic partitioning, the compiler cannot determine the precise relationship of all array operands at compile time--since subarrays may overlap or may not--depending upon the dynamics of the situation.

The first problem may be overcome by noting that the expression tree generated by the parsing phase of the OL/2 compiler reflects the complete structure of the array expression. The second problem, which involves dynamic partitioning [1,9], allows one to partition arrays and define subarrays. The exact structure of the partitions can change dynamically at run time so that the sizes, shapes and relations of the subarrays may change dynamically. The most difficult problem that arises from partitioning, as far as compilation is

concerned, is the possible overlap of the subarrays. For instance, if an array is partitioned with two independent partitions, then the corresponding subarrays may overlap and the subarrays have elements in common. Since the subarrays may be referenced with independent identifiers, that is independent of the original array identifier, the compiler must look at the underlying structure which controls partitioning. The compiler is clearly limited to the static aspects of the problem. Several examples will be considered to illustrate these points.

If A, B, and C do not overlap, then no temporary variables need be generated in the array assignment

$$A = B + C$$

since the result of adding B and C can be accumulated directly into the storage for A, since A must be of the same dimensionality and size as the result of the operation B + C. If overlap occurs between A, B, or C, then one temporary variable must be generated, since accumulating B + C directly into A may overwrite some of the elements of A.

Next, consider the array assignment and assume that the subarrays do not overlap.

$$A = B * C + D + E$$

In this case the order of evaluation becomes important. D + E can be evaluated first and accumulated into the storage for A. Then a temporary array must be created for B * C, call it TEMP21. Since operands are always combined in a binary manner, B * C cannot be put in A, since A has already been used to store D + E. But, if B * C is evaluated first and put directly into the storage for A, then D and E can be accumulated into the B * C result, one at a time, requiring no temporary variables. With subarrays of overlap the above analysis becomes much more complicated.

The problems considered in the above examples become more involved

when one considers OL/2 "geometric" data types. For example, a tridiagonal matrix can be added to a rectangular matrix if both matrices have the same number of rows and columns, where nonexistent elements in the tridiagonal matrix, which are not stored, are implicitly defined as zero. The result is a rectangular matrix. Also, later versions of OL/2 will allow arrays to be real, complex, integer, or rational, and different combinations of operands will require different temporary storage sizes.

The examples given suggest that the following points be considered in the design of an algorithm that is to produce a minimum temporary variable evaluation for an array expression.

1. The problem of overlap cannot be completely determined at compile time, due to dynamic partitioning, but some cases can be determined from the information structure that controls dynamic partitioning and some overlap cases definitely be eliminated.
2. The ordering of evaluations of expressions is important in relation to the number of temporary variables generated.
3. Certain array size relations can be utilized. For example, subarrays require at most the same amount of storage as the parent array from which they are derived.

Several criteria have been mentioned which are necessary for an algorithm that is to operate on an OL/2 expression tree to minimize temporary variables. One approach is to create a threaded tree and let the threads indicate the continuing scope of all possible temporary variables. However, further work is required to assure that all possible information available at compile time is utilized by the compiler.

Another approach to temporary variable handling would be to defer all decisions as to temporary generation until run time when all size and overlap information is known. The drawback of this technique is that essentially the coding of OL/2 expressions would be deferred until run time with the optimized expression tree being passed to run time procedures. This technique would give optimal results in reducing temporary variable storage but might increase the execution time of a program considerably if a given expression to be coded were contained in a loop, but for very large arrays this extra computation time might be insignificant.

Currently, the reuse of temporary variables that have already been generated is handled by the OL/2 run-time procedures. For example, in the expression

$$A = B + C + D$$

a temporary, call it TEMP21, would be generated for the result of C + D. B + TEMP21 is then accumulated into the storage available for TEMP21, assuming the storage size of TEMP21 is large enough for the various geometric types, TEMP21 could then be assigned to A to complete the evaluation. The reuse of temporary variables is only a subgoal of the general problem outlined above, and the storage allocated to A has not been used because of possible overlap.

There are many problems which must be solved when considering the various possibilities that can arise in evaluating array expressions in language such as OL/2. Several important aspects of the problem associated with the generation of temporary arrays have been covered, but further work is required in this area.

5. CODING PHASE

5.1 Description of the Code Produced

The task of the coder is to transform the intermediate code, in the form of the concrete expression tree, into "object" code that can be executed at run time. In OL/2, the object code produced consists of a series of call statements to assembly language operational routines, that with the proper information passed, can carry out the necessary matrix operations efficiently at run time. Essentially the coder transforms the expression tree into a linear list of object code calls that, when executed sequentially, produce the necessary expression evaluation. The object code is effectively a simulated parallel execution of array operations.

The format of the object code is fairly flexible as far as the compiler is concerned and has the following properties.

1. The operation to be performed on the operands passed is specified by the name of the routine called.
2. The two operands are passed as arguments to the operational routines as well as the name of the variable or temporary variable that is to be the result of the operation.
3. The operational routines handle all types of arrays, all geometric shapes, and all combinations. For example, a lower triangular matrix times a rectangular matrix is specified as a matrix times matrix operation with no special consideration given to the geometrical structure, since this is a task which is handled by the routine itself with the help of other information structures.

4. Storage for temporary results is created and deleted at run time by the operational routines under the control of storage flags passed as arguments to the routines. The reuse of temporary variables, if possible, is handled by the run time routines.
5. Negation and transposition of the operands and temporary results can be specified either before or after the operation is to be carried out. This is accomplished through the negate and transpose flags that are passed as arguments to the operational routines.
6. Sequence number computations are done in the operational routines, that is, sequences of arrays and the designation of a specific array in the sequence is handled within the routine itself.
7. Null operands are also handled in the operational routines, (see [7]).

Example call sequences generated for several OL/2 statements are shown in Table 5.1. It is interesting to note that such a set of calling statements could serve as the instruction set for an array processing computer system as indicated in [4].

In Table 5.1, it can be noted that temporary variable names must be created by the coder to represent temporary computational results, and that the storage allocation of the associated temporaries is under control of the coder generator through the storage allocation bits in the instruction calls.

The handling of scalar expressions in the coder is delegated to the PL/1 compiler which compiles the object code for OL/2. That is, although scalar expressions are parsed into a precedence tree format, they are reassembled by

(a) EXPRESSION: $A = \alpha * B + C$

CODE:

```
Call @OLSMO (α, 0, 0, 0, 0, $B1, 1, 0, 0, 0, $TEMP21);
Call @OLOPS (0, 0, 0, 0, $TEMP21, 1, 0, 0, 0, $C1, 1,
        0, 0, 0, $TEMP22);
Call @OLOPS (0, 0, 0, 0, $TEMP22, 0, 0, 0, 0, $OL2NULL,
        0, 1, 0, 0, $A1);
```

(not a minimum temporary variable evaluation, see section 4)

(b) EXPRESSION $\beta = (x,y) * || - A ||$

CODE:

```
Call @OLINP (1, 0, 0, 0, $x1, 1, 0, 0, 0, $y1, #TEMP01);
Call @OLNORM (1, 1, 0, 0, $A1, #TEMP02); β = #TEMP01 *
        #TEMP02;
```

(c) EXPRESSION: $A' = x' * \alpha * \beta * \gamma * (-B)$

CODE:

```
Call @OLSMO (α * β * γ, 1, 0, 1, 0, $x1, 1, 0, 0, 0,
        $TEMP11);
Call @OLOMO (0, 0, 0, 0, $TEMP11, 1, 1, 0, 0, $B1,
        1, 0, 1, 0, $A1);
```

(this is a minimum operation and temporary variable evaluation)

Table 5.1. 01/2 Expressions and Code

the coder into expression strings and passed as a single argument in an operational routine calling statement. For example, in Table 5.1(c), the expression and its associated code illustrates this property.

The information passed to the run-time operational routines is contained in the parameter values in the compiler generated call statements. The routines utilized to carry out the basic OL/2 operations are as follows:

@OLSMO - carries out ARRAY-SCALAR * and VEC-SCALAR * operations.

@OLOMO - carries out ARRAY * and ARRAY-VEC * operations.

@OLOPS - carries out ARRAY SUM operation and A = A operation.

@OLINP - carries out INNER PROD operation.

@OLNORM - carries out NORM operation.

The parameter information contained in the operational calls can be defined by the following names: NODE_POINTER, SEQUENCE=#, STORAGE_BIT, NEGATE_TAG, and TRANSPOSE_TAG.

NODE_POINTER is a pointer variable that points to a run-time dope vector that describes the storage format for an operand or a result. Pointers always begin with the character $, followed by the OL/2 name of the variable declared, followed by the OL/2 block # in which the variable was declared. For example, $A1 is the name given to the run time pointer variable that points to the dop vector for the OL/2 variable A declared in block 1 of the OL/2 source code.

SEQUENCE_# is used for sequences of arrays or scalars. A zero value indicates no sequence, or the first element of a declared sequence.

STORAGE_BIT is a flag that is used for dynamic storage allocation. For operands, "0" indicates that the run-time storage routines can free the run-time storage for the associated variable after the operation is completed.

A "1" indicates that the variable is to be used after the operation is completed and that the run-time variable contents should be left intact. For results, a "0" indicates that storage is not required for a temporary result; a "1" indicates that storage should be created at run time for a temporary result.

A NEGATE_TAG that is "1" indicates that the array operand should be negated before doing the operation, (actually the elements of the array are negated as the operation proceeds at run time). For an array result, a "1" means that the result should be negated after the operation. "0" means no negation.

A TRANSPOSE_TAG that is "1" indicates that an operand should be accessed as a transpose, or the result stored as a transpose. "0" indicates that no transposition need be done.

The calling formats for the operational routines can be illustrated using the parameter definitions above. Define an operand or result parameter sequence as follows:

PARM_SEQUENCE = STORAGE_BIT, NEGATE_TAG, TRANSPOSE_TAG,
SEQUENCE_#, NODE_POINTER

Then the calling formats are:

@OLSMO - Call @OLSMO (SCALAR_EXPRESSION_OPERAND, PARM_SEQUENCE for operand, PARM_SEQUENCE for result);

@OLOMO, @OLOPS - Call @OLOMO (
@OLOPS (PARM_SEQUENCE for first operand, PARM_SEQUENCE for second operand, PARM_SEQUENCE for result);

@OLINP - Call @OLINP (PARM_SEQUENCE for first operand, PARM_SEQUENCE for second operand, SCALAR_RESULT);

@OLNORM - Call @OLNORM (PARM_SEQUENCE for operand, SCALAR_RESULT);

In the case of A = A, the second operand is filled in as an OL/2 null operand

in a call to @OLOPS, since a null operand follows the rule $A = \emptyset + A$, so that $A = \emptyset + B$ is equivalent to the assignment $A = B$.

Naming conventions are followed for temporary variable names generated by the compiler. Variables that begin with $TEMP are node pointers to run time dope vectors of array variables. Variables that begin with #TEMP are the variable names corresponding to a scalar temporary value at run time.

5.2 The Coding Algorithm

Basically, the coding algorithm traverses the expression tree, using a modified form of the right end order traveral algorithm T of Knuth [6]. When an operator node is visited in the traversal, the proper parameters for the OL/2 run time object code call are built up, if necessary, in the variable CODE_STRING. After the visit is completed, a completely coded call statement will exist in CODE_STRING that represents the result of the operation being considered. The coding algorithm for the basic OL/2 operations, given a properly constructed expression tree, is given below. Coding for the OL/2 operations not mentioned is easily added to the given algorithm, although for some operations many special cases must be considered. The coding algorithm for the basic OL/2 operations is as follows:

CA1: Traverse the expression tree using algorithm T from Knuth modified for an endorder traversal. The algorithm terminates with a pointer to the next node to be visited in an endorder traversal.

CA2: If the node to be visited is a leaf (operand node) then go to CA1 (no processing need be done); otherwise, go to CA3 (an operator node has been found).

CA3: Set CODE_STRING array members to the null string.

CA4: Transfer to the appropriate section using the operator type contained in the operator node; go to CA5 for operator type = INNER PROD, or ARRAY-VEC *, or ARRAY *, or VEC-SCALAR *, or ARRAY-SCALAR *, or ARRAY SUM, or NORM, or A = A or A = S; go to CA6 for operator type = ARRAY or TRANS; go to CA7 for operator type = S * S, S + S, S - S, S/S, S**S, S = S.

CA5: Process an operation involving at least one array. Do for right and left operand nodes below the operator if present and not scalar operands;

51: Reduce #_OF_TIMES_TO_USE of operand node field by one.

52: If the storage for the operand can now be deleted, i.e., it is a temporary variable that is no longer needed, set STORAGE_BIT in CODE_STRING for this operand to '0'; otherwise set STORAGE_BIT in CODE_STRING to '1' to indicate that the operand is to be saved.

53: Put NEGATE_TAG and TRANSPOSE_TAG for the operand node in the proper position in the code string.

54: Put SEQ_# associated with the operand (may be zero indicating a non-sequenced variable) in the proper CODE_STRING position.

End DO loop. Copy any scalar operands as strings into proper CODE_STRING position. Go to CA8.

CA6: Process a unary transpose or negate array operation.

1: Copy the operand node information into the operator node.

2: Complement the negate or transpose field of the operator node.

Set link pointer in operand node to NULL value (making the operator node into a leaf operand node), and go to CA1 (no code need be generated).

CA7: Process an operation involving only scalar subexpressions.

71: Concatenate left and right operand scalar strings together with proper operator symbol in between and place in the operator node.

72: Set link pointers of the operand node to the NULL value (making the operator node into a result operand leaf node).

Go to CA1 (no code need be generated).

CA8: Set up name and code parameters for the temporary result.

81: Get a name for the temporary result of the operation from the name pool and insert it in CODE_STRING.

82: Set the STORAGE_BIT for the result variable to '1' in CODE_STRING indicating that the run time routines must get storage for the result of the given operation.

CA9: Insert the proper name into the CODE_STRING through a table lookup, of the operational routine that is to be called at run time.

CA10: Set the link pointers of the operator node to the value NULL, making this node now a leaf operand node.

CA11: Save the contents of CODE_STRING, which is now the complete OL/2 run time code for the operation being considered.

CA12: Go to CA1 to visit the next node in the endorder sequence.

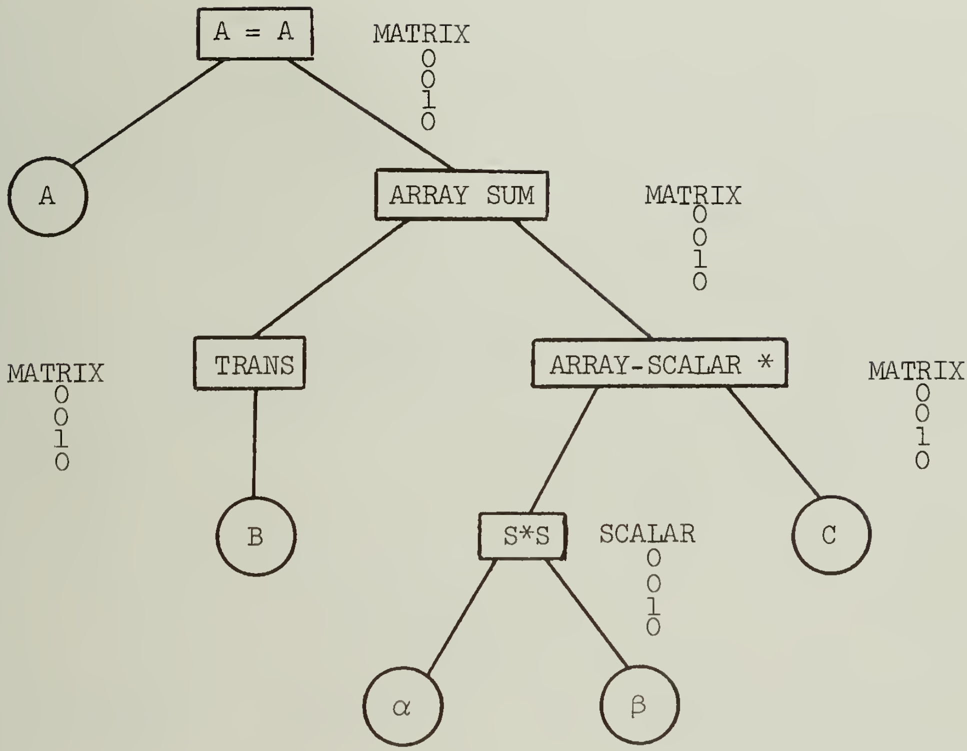
5.3 Example of Code Generation

As an example of the application of the coding algorithm, consider the expression tree given in Figure 5.2. The information that would be put into the nodes of the expression tree by the tree building algorithm is given in the figure. The following steps would take place in coding the given tree:

1. The tree is traversed in right endorder, with the C node being visited. No action is taken since this is a leaf node.
2. β is visited with no action taken, α is visited with no action taken.
3. The node $\boxed{\text{S * S}}$ is visited, and since it is an operator node, (non-leaf node), involving only scalar operands, it is modified as in Figure 5.3(a). '$\alpha * \beta$' becomes a scalar string result of $\boxed{\text{S * S}}$ with no code being generated.
4. $\boxed{\text{ARRAY SCALAR *}}$ is visited, and since it is a binary operator node of a non-scalar type, code is generated and the tree modified as in Figure 5.3(b). Notice that a temporary variable name is established for the array result of the operation. TEMP21 indicates that it is the first variable name generated for a temporary of dimension 2.
5. The leaf node B is visited with no action being taken.

EXPRESSION: $A = B' + (\alpha * \beta) * C$

EXPRESSION TREE:



NODE FIELDS GIVEN:

TYPE_EXP

TRANSPOSE_TAG

NEGATAGE_TAG

#_OF_TIMES_TO_USE

SEQUENCE_#

Figure 5.2. Example to be Coded

6. The operator node $\boxed{\text{TRANS}}$ is visited. No code need be generated for the transpose operation at this time, the tree changes are given in Figure 5.3(c).

7. The $\boxed{\text{ARRAY SUM}}$ operator node is visited, code is generated and the tree modified as in Figure 5.3(d). Notice that the transpose bit for the operand B in the call statement is set, indicating that the run time operational routine @OLOPS should access the operand B in transposed order when B is used in the sum. The name TEMP22 is generated for the result, and the result storage bit is set to one, indicating that run time storage for the result must be found. The run time routines will reuse the storage allocated for TEMP21 if possible, since it is designated in the code as "free storage", by a zero storage bit, after the operation.

8. The leaf node A is visited with no processing done.

9. The root node of the tree $\boxed{\text{A = A}}$ is visited, and code is generated for the assignment. Note that in the coding process, (see CA5:51 in coding algorithm), #_OF_TIMES_TO_USE is reduced to zero for the node TEMP22. The storage bit for TEMP22 in the OL/2 code call to @OLAASS is set to zero, indicating that at run time after executing the assignment operation the storage for TEMP22 can be deleted. (See Figure 5.3(e)).

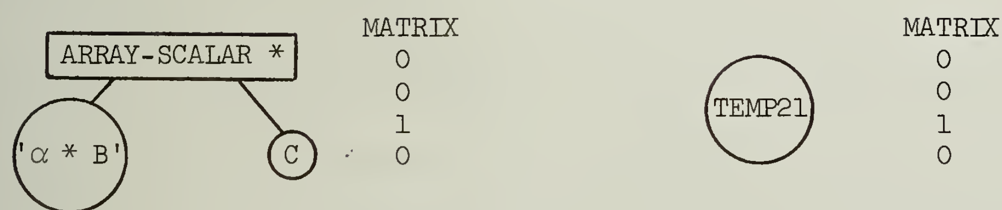
At this point the entire tree has been traversed and is, therefore, coded. The complete code generated for run time execution is then as follows:

```
Call  @OLSMO (α * B, 1, 0, 0, 0, $C1, 1, 0, 0, 0, $TEMP21);
Call  @OLOPS (1, 0, 1, 0, $B1, 0, 0, 0, 0, $TEMP21, 1, 0, 0,
                    0, $TEMP22);
```

(a)

No code generated

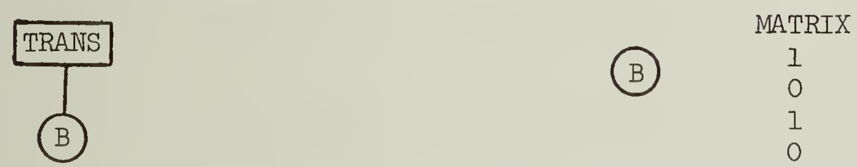
(b)



Code generated:

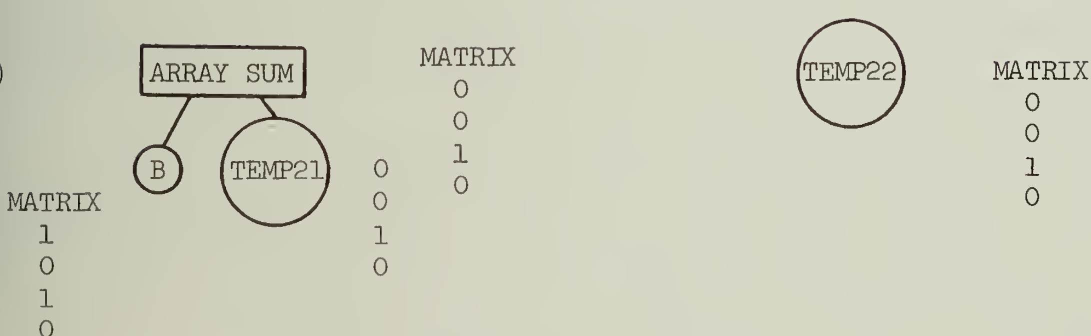
Call @OLSMO (α * β, 1, 0, 0, 0, $ C1, 1, 0, 0, 0, $TEMP21);

(c)



No code generated

(d)

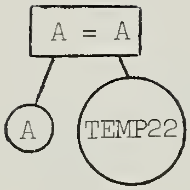


Code generated:

Call @OLOPS (1, 0, 1, 0, $B1, 0, 0, 0, 0, $TEMP21, 1, 0, 0, 0, $TEMP22);

Figure 5.3. Tree Changes in Coding

(e)



MATRIX
0
0
1
0

MATRIX
0
0
1
0

A

Code generated:

```
Call  OLOPS (0, 0, 0, 0, $TEMP22, 0, 0, 0, 0, $OL2NULL
             1, 0, 0, 0, $A1);
```

Note: Node fields given are listed as follows:

TYPE_EXP
TRANSPOSE_TAG
NEGATE_TAG
#_OF_TIMES_TO_USE
SEQUENCE_#

Figure 5.3 (continued). Tree Changes in Coding

```
Call  @OLOPS (0, 0, 0, 0, $TEMP22, 1, 0, 0, 0, $OL2NULL, 0, 0,
              0, 0, $A1);
```

It is interesting to note that, as mentioned in section 4, the complete OL/2 code for this expression could be as follows:

```
Call  @OLSMO (α * B, 1, 0, 0, 0, $C1, 0, 0, 0, 0, $A1);
Call  @OLOPS (1, 0, 1, 0, $B1, 1, 0, 0, 0, $OL2NULL, 0, 0, 0,
              0, $A1).
```

This coding of the expression requires no temporary variables and one less OL/2 run time call, assuming that none of the operands overlap in storage.

6. RESULTS AND EXAMPLES

The algorithms described in sections 3 and 5 (the expression tree builder, and coding algorithm) have been implemented and are currently part of the OL/2 compiler. Appendix A contains the syntactic specification of the current version of OL/2, while Appendix B contains the PL/1 source listing of the associated semantic action routines for the expression processing module of the OL/2 compiler.

Appendix C contains examples of the code generated by the compiler for various OL/2 input source expressions. It can be noted that in each example a minimum operation evaluation occurs and that the storage associated with all temporary array variables generated by the compiler is reused at run time whenever this is possible.

LIST OF REFERENCES

[1] H. C. Adams, "Dynamic Partitioning in the Array Language OL/2," M.S. Thesis, University of Illinois at Urbana-Champaign, Department of Computer Science Report No. 421, January 1971.

[2] J. L. Gaffney, Jr., "TACOS: A Table Driven Compiler-Compiler System," M.S. Thesis, University of Illinois at Urbana-Champaign, Department of Computer Science Report No. 352, June 1969.

[3] F. R. A. Hopgood, Compiling Techniques, Elsevier, Amsterdam, Netherlands, 1969.

[4] D. R. Jurich, "An Asychronous Modular Parallel Pipelined Multiprogrammed Array Assembly Language Computer Organization," Class Term Paper CS 465, University of Illinois at Urbana-Champaign, Department of Computer Science, June 1972.

[5] J. L. Latch, "An Algorithm for Evaluating Array Expressions in OL/2," M.S. Thesis, University of Illinois at Urbana-Champaign, Department of Computer Science Report No. 422, January 1971.

[6] D. E. Knuth, The Art of Computer Programming Fundamental Algorithms, Vol. 1, Addison Wesley, Reading, Mass., 1969.

[7] J. R. Phillips, "The Structure and Design Philosophy of OL/2--An Array Language--Part I: Language Overview," University of Illinois at Urbana-Champaign, Department of Computer Science Report No. 544, September 1972.

[8] J. R. Phillips, "The Structure and Design Philosophy of OL/2--An Array Language--Part II: Algorithms for Dynamic Partitioning," University of Illinois at Urbana-Champaign, Department of Computer Science Report No. 420, September 1971.

[9] J. R. Phillips, and H. C. Adams, "Dynamic Partitioning for Array Languages," to appear in CACM, 1972.

APPENDIX A

OL/2 EXPRESSION SYNTAX AND THE TACOS SYSTEM

The compiler for OL/2 is generated by a general compiler-compiler system called TACOS, TAable driven COmpiler-compiler System [2]. TACOS is designed to accept the input syntax of a language and the associated semantic specification, and to produce a linked list syntax table and semantic action procedure that completely specifies the input form of the language. A fixed interpretive top-down parser-scanner module then operates on the TACOS produced specification to provide compilation facilities for the input language.

The input syntax specification for a language to the TACOS system is written in IBNF, Interpretable Barus Naur Form, which is similar to the standard BNF specification for a language. IBNF is an extension of BNF, which simplifies the syntax specification and the table building task of the TACOS system. The extensions of BNF incorporated into IBNF are:

1. Parenthetical expressions are permitted in order to reduce the number of phrase classes.
2. Three repetition characters are defined in order to allow a simplified syntactical specification and to eliminate left recursion problems in table building, (see Table A1).
3. Certain intrinsic terminal phrase class symbols are defined to provide for the more efficient processing of terminal symbols, (see Table A2).

If any of the tests in TABLE A2 fail, for example, the string to be extracted is not found, the parser scanner variable OK is set to zero indicating that the parser scanner must back up and try again in its top down analysis. If the test succeeds OK is set to the value one, and syntax analysis along the

Repetition Character	Number of Occurrences	IBNF Example	BNF Interpretation
+	≥ 1	< A >::=< B > +	< A >::=< B >|< A > < B >
*	≥ 0	< A >::=< B > *	< A >::=< empty >|< A > < B >
?	0 or 1	< A >::=< B > ?	< A >::=< empty >|< B >

Table A.1. IBNF Repetition Symbols

IBNF Intrinsic Terminal Symbol	Meaning	Example
< * I >	Test for and extract an identifier from the input string, if present, and put it in the string variable TEMPIDENT	< * I >
< * N >	Test for and extract an integer number from the input string, if present, and put it in the string variable TEMPCONST	< * N >
< # N >	Go to the semantic action routine labelled ACTION-n at this point in the parse	< # 5 > go to ACTION-5
.KEYWORD.	Test for and extract the keyword enclosed in the periods, if present, in the input string and put it in the string variable TEMPSTRING	.PROCEDURE.
'DELIMITER'	Test for and extract the delimiter string within the quotes from the input string, if present, and put it in the string variable TEMPSTRING	

Table A.2. IBNF Intrinsic Terminal Symbols

current analysis path continues. The variable OK can be set in the semantic action routines indicating that the phrase class "test" $< \#n >$ fails.

The semantic action routines in TACOS are written in PL/1, and contained in the external procedure ACT. The semantic routines have access to the input source code through the character string variable CHAR. Systems procedures CARDNUM and CARDCOL can be called by the semantics, and return values that indicate the farthest point in the input source string which the parser scanner has reached. These features can be used for the isolation of source language errors. Additional details of the TACOS system may be found in Gaffney [2].

The following pages of this appendix contain the IBNF syntactical description of OL/2 array expressions. Appendix B contains the semantic action specification for OL/2 array expressions.

```
/*********************************************************************/
/*                                                                   */
/*                                                                   */
/*        OL/2 ARRAY EXPRESSION SYNTAX.                              */
/*                                                                   */
/*                                                                   */
/*********************************************************************/


<ASSIGNMENT_STATEMENT> ::= <ASSIGN_SCAN> ( <OL2_LEFT_HAND_SIDE> ( ','
          <OL2_LEFT_HAND_SIDE> )* ( '=' | '<-' )
          ( ( '@' | '"NULL"' ) <#46> | <OL2_ARITHMETIC_EXPRESSION>
          <#42> <#41> ) ';' | <#43> ) ;


<ASSIGN_SCAN> ::= <*I> <#8> ;


<OL2_LEFT_HAND_SIDE> ::= <OL2_IDENTIFIER>
          ( '|_' <#104> '_|' <#22> )? ( ( '<' <#113> '>' )+ <#44> )?
        ( '(' <#70> ')' <#45> )?    | ( <#24> <REFERENCE> <#25> |
          <#40> )                                                    ;


<OL2_ARITHMETIC_EXPRESSION> ::= <OL2_TERM> ( ( '+' <OL2_TERM> <#10>
          | '-' <OL2_TERM> <#11> ) <#12> )*                          ;


<OL2_TERM> ::= <OL2_DIVIDE> <#13> ( '*' <OL2_DIVIDE> <#14> <#16>
          )* <#17>                                                   ;


<OL2_DIVIDE> ::= <OL2_FACTOR> ( '/' ( <MODIFIED_EXPRESSION_UNIT>
           | <EXTENDED_SCALAR_EXPRESSION> |
           <MODIFIED_OL2_IDENTIFIER> ) <#15> <#16> )* ;


<OL2_FACTOR> ::= <OL2_PRIMARY> ( '**' <OL2_EXTRA> <#18> )?           ;


<OL2_EXTRA> ::= ( <MODIFIED_EXPRESSION_UNIT> |
           <EXTENDED_SCALAR_EXPRESSION> ) ( '**' <OL2_EXTRA>
           <#18> )? ;
```

```
<OL2_PRIMARY> ::= <MODIFIED_EXPRESSION_UNIT> |
            <EXTENDED_SCALAR_EXPRESSION> | <MODIFIED_OL2_IDENTIFIER>
            ;

<MODIFIED_EXPRESSION_UNIT> ::= ('+' | '-' <#155> )? '(' <#20>
            <OL2_ARITHMETIC_EXPRESSION> ')' <#19> ('''' <#21> )? ;

<MODIFIED_OL2_IDENTIFIER> ::= ('+' | '-' <#155> )? <OL2_IDENTIFIER>
            <#19> ( '|_' <#104> '_|' <#22> )?
          ( ( '<' <#113> '>' )+ <#44> )? ( '''' <#21> )?   ;

<OL2_IDENTIFIER> ::= <*I> <#23>                                    ;

<EXTENDED_SCALAR_EXPRESSION> ::= <#24> <PL1_AND_OL2_SCALARS> <#25> |
            <#40> ;

<PL1_AND_OL2_SCALARS> ::= ( '+' | '-' <#155> )? <BASICS> <#19> ;

<BASICS> ::= '(' <#20> <PL1_AND_OL2_SCALARS> ')' | <INNER_PRODUCT> |
            <NORM>          | <REFERENCE> | <CONSTANT>              ;

<REFERENCE> ::= <BASIC_REF> ( '->' <BASIC_REF> )*                  ;

<BASIC_REF> ::= <UNQUAL> ( '.' <UNQUAL> )*                         ;

<UNQUAL> ::= <#26> <*I> <#39> ( '|_' <#104> '_|' )? ( '''' <#156> |
            <#157> ) ( '<' <#113> '>' <#158> )? ( '(' <#27>
          <OL2_ARITHMETIC_EXPRESSION> <#28> ( ','
          <OL2_ARITHMETIC_EXPRESSION> <#28> <#29> )* ')' <#30>
           | <#31> ) <#38>                                         ;

<CONSTANT> ::= <#32> ( <*N> )? ( '.' )? ( <*N> )? <#33>
          ( 'E' ( '+' | '-' )? <*N> ( 'I' )? )?   ;
```

```
<NORM> ::= '||'        <OL2_ARITHMETIC_EXPRESSION> '||' <#34>          ;

<INNER_PRODUCT> ::= '('        <OL2_ARITHMETIC_EXPRESSION> <#35>
          ',' <OL2_ARITHMETIC_EXPRESSION> <#35> ')' <#36>         ;



<OL2_BOOLEAN_EXP> ::= <#163> <B1> ( '|' <B1> <#159> )* ;


<B1> ::= <B2> ( '&' <B2> <#160> )* ;


<B2> ::= '(' <#161> <B2> ')' | '¬' <B2> <#162> |
                <COMPARE_EXPRESSION> (
                <COMPARE_OP> <COMPARE_EXPRESSION> <#164> | <#167> ) ;


<COMPARE_OP> ::= ( '¬=' | '¬>' | '¬<' | '>=' | '<=' ) <#165> |
                ( '=' | '>' | '<' ) <#166> ;


<COMPARE_EXPRESSION> ::= ('@' | '"NULL=' ) <#46> |
                <OL2_ARITHMETIC_EXPRESSION> ;
```

APPENDIX B

```
/*************************************************************************/
/*                                                                       */
/*                                                                       */
/*                                                                       */
/*       OL/2 SEMANTIC ACTION ROUTINES FOR EXPRESSION HANDLING.          */
/*                                                                       */
/*                                                                       */
/*                                                                       */
/*************************************************************************/




ACT:  /*  OL/2 SEMANTIC ACTION ROUTINE PROCEDURE  */
      PROCEDURE(WHICH_ACTION_NUM) RECURSIVE;

/*************************************************************************/
/*                                                                       */
/*       THE FOLLOWING DECLARATION STATEMENTS REFER TO VARIABLES         */
/*       THAT ARE COMMOM TO THE PARSER SCANNER MODULE AND THE            */
/*       SEMANTIC ACTION ROUTINES.                                       */
/*                                                                       */
/*************************************************************************/


       DCL WHICH_ACTION_NUM FIXED BIN (31,0) ,
         1 BRIDGE EXTERNAL,
         2 GOCONDITION FIXED BIN (31,0) ,
         2 TEMPCONST VARYING CHARACTER(15),
         2 TEMPIDENT VARYING CHARACTER(32),
         2 TEMPSTRING VARYING CHARACTER(100),
               CHAR ENTRY (FIXED BIN(31)) RETURNS (CHAR(1)),
               ACHAR ENTRY (FIXED BIN(31),CHAR(1)),
           (OK , INP) FIXED BINARY (31,0) EXTERNAL ,
           CARDNUM EXTERNAL ENTRY RETURNS(FIXED BINARY (31,0)) ,
           CARDCOL EXTERNAL ENTRY RETURNS(FIXED BINARY (31,0)) ,
               ACTION(0:220) LABEL STATIC;
         DECLARE FARTHEST FIXED BIN(31,0) EXTERNAL INITIAL(1) ;
 DCL MOVCHAR ENTRY (,FIXED BIN(31),FIXED BIN(31));
```

```
/***************************************************************************/
/*                                                                         */
/*       THE VARIABLES BELOW ARE INITIALIZED AS INDICATED, THE            */
/*       FIRST TIME THAT THE PROCEDURE ACT IS TRANSFERED TO.              */
/*                                                                         */
/***************************************************************************/

         ACTION_12_ROUTINE(-1 )=ACTION_12_ROUTINE_PLUS;
         ACTION_12_ROUTINE(-2 )=ACTION_12_ROUTINE_MINUS;
         ACTION_12_ROUTINE(-3 )=ACTION_12_ROUTINE_DIVIDE;
         ACTION_12_ROUTINE(-4)=ACTION_12_ROUTINE_MULTIPLY;
         ACTION_12_ROUTINE(-5)=ACTION_12_ROUTINE_EXPONENTIATE;
         ACTION_12_ROUTINE(-6)=ACTION_12_ROUTINE_INNER_PRODUCT;
         ACTION_12_ROUTINE(-7)=ACTION_12_ROUTINE_EQUAL;
         ACTION_12_ROUTINE(-8)=ACTION_12_ROUTINE_NORM;
         ACTION_12_ROUTINE(-9)=ACTION_12_ROUTINE_UNIMINUS;
         ACTION_12_ROUTINE(-10)=ACTION_12_ROUTINE_TRANSPOSE;
         ACTION_12_ROUTINE(-11)=ACTION_12_ROUTINE_SEQUENCE;
         ACTION_12_ROUTINE(-12)=ACTION_12_ROUTINE_MODIFY ;
         ACTION_12_ROUTINE(-13)=ACTION_12_ROUTINE_SEPARATOR ;
         ACTION_12_ROUTINE(-14)=ACTION_12_ROUTINE_FUNCTION ;
         ACTION_12_ROUTINE(-15)=ACTION_12_ROUTINE_CONCATENATE;
         ACTION_12_ROUTINE(-16)=ACTION_12_ROUTINE_LHS_E_E ;
         ACTION_12_ROUTINE(-17)=ACTION_12_ROUTINE_PART_OF ;
         ACTION_12_ROUTINE(-18)=ACTION_12_ROUTINE_OL2_COMPARE;
         ACTION_12_ROUTINE(-19) = ACTION_12_ROUTINE_BOOL_OP ;
         ACTION_12_ROUTINE(-20) = ACTION_12_ROUTINE_NOT_OP ;


         ALLOCATE TREE_NODE ;
         RLINK , LLINK = NULL ;
         STRING_POINTER = ADDR(OL2NULL_DUMMY) ;
         #_OF_TIMES_TO_USE = UNDEFINED ;
         $TYPE_CODE = 0 ;
         $#DIMENSIONS = UNDEFINED ;
         NEGATE_TAG , TRANSPOSE_TAG , IDENTITY_TAG = NO ;
         $OL2NULL = NODE_POINTER ;

/***************************************************************************/
/*                                                                         */
/*       GO TO THE PROPER ACTION ROUTINE; THE ONE SPECIFIED IN            */
/*       SYNYAX.                                                           */
/*                                                                         */
/***************************************************************************/

      CALL GOTO( ACTION(WHICH_ACTION_NUM) ) ;
```

```
/*************************************************************************/
/*                                                                       */
/*      THE FOLLOWING ARE DECLARATIONS FOR SUBROUTINES USED              */
/*      BY THE EXPRESSION HANDLING MODULE OF THE SEMANTICS.              */
/*                                                                       */
/*************************************************************************/
 DCL CODER_#_1 ENTRY(POINTER);
 DCL ACT ENTRY (FIXED BINARY (31,0)) ;
 DCL SEARCH ENTRY ( CHAR(32) VARYING , FIXED BIN (31,0) )
         RETURNS ( POINTER )                                             ;
 POINTER_TO_STRING ENTRY (CHAR (*) VARYING,AREA(*)) RETURNS (POINTER);
 DCL BUILD_AND_STACK_SCALAR_NODE ENTRY (FIXED BIN (31,0) ,
         FIXED BIN (31,0) )                                              ;
 DCL BUILD_AND_STACK_STRING_NODE ENTRY ( CHAR(100) VARYING )             ;
 DCL #ERROR ENTRY ( FIXED BINARY (15,0) )                                ;
 DCL IS_AN_OL2_ENTRY ENTRY ( CHAR(32) VARYING ) RETURNS ( BIT(1) )       ;

/*************************************************************************/
/*                                                                       */
/*      EXPRESSION PARSING PRECEDENCE MATRIX.                            */
/*                                                                       */
/*************************************************************************/

         DCL PRECEDENCE_TABLE (0:4,0:4) FIXED (15,0)
          INITIAL (0,1,1,1,1, 1,1,1,1,1, 1,1,2,1,2, 1,1,0,1,0,
          1,1,0,1,1) ;

/*************************************************************************/
/*                                                                       */
/*      THE STRUCTURE DECLARATION BELOW IS FOR THE OPERAND AND           */
/*      RESULT NODES.                                                    */
/*                                                                       */
/*************************************************************************/

       DECLARE
          1 TREE_NODE BASED (NODE_POINTER) ,
           ( 2 RLINK  ,
             2 LLINK  ,
             2 SEQ_#_PTR  ,
             2 STRING_POINTER ) POINTER,
           ( 2 $#DIMENSIONS,
             2 PART_SIZE   ,
             2 #_OF_TIMES_TO_USE   ,
             2 TYPE_EXP  ,
             2 $TYPE_CODE ) FIXED BINARY (15,0) ,
           ( 2 NEGATE_TAG ,
             2 TRANSPOSE_TAG ,
             2 IDENTITY_TAG ) BIT (1) ;
```

```
/**********************************************************************/
/*                                                                    */
/*       THE DATA STRUCTURES UTILIZED IN REPEATED SUBEXPRESSION       */
/*       HANDLING ARE DECLARED BELOW.                                 */
/*                                                                    */
/**********************************************************************/

        DCL  1 SUBTREE_TABLE (50) ,
               ( 2 POINTER_TO_OP1 ,
                 2 POINTER_TO_OP2 ,
                 2 POINTER_TO_RESULT ) POINTER ,
                 2 OPERATION FIXED BIN (15,0) ,

             CURRENT_TABLE_PTR FIXED BIN(15,0)  INITIAL(0) ;


/**********************************************************************/
/*                                                                    */
/*       THE DECLARATIONS BELOW ARE FOR MISCELLANEOUS VARIABLES       */
/*       REFERENCED BY THE EXPRESSION HANDLING MODULE.                */
/*                                                                    */
/**********************************************************************/


DECLARE  (
                OL2_ENTRY INITIAL (0) , OL2_ID INITIAL (1) , OTHER
                INITIAL (2),
                CURRENT_OP , SUBTREE_TYPE(50)
                , SCALAR INITIAL (0) , FUNCTION INITIAL (1)
                ,COL_VEC INITIAL (2) , ROW_VEC INITIAL (3)
                , MATRIX INITIAL (4) , PLUS INITIAL (-1) ,
                MINUS INITIAL (-2) , DIVIDE INITIAL (-3) ,
                MULTIPLY INITIAL (-4) , INNER_PRODUCT INITIAL (-6)
                , EXPONENTIATE INITIAL (-5) , NORM INITIAL (-8)
                , UNIMINUS INITIAL (-9) , TRANSPOSE INITIAL (-10)
                , CONCATENATE INITIAL (-15) , SEQUENCE INITIAL(-11)
                , EQUAL INITIAL (-07) , SEPARATOR INITIAL (-13)
                , FUNCTION_TYPE INITIAL (-14) ,
                MODIFY_TO_TYPE_PL1 INITIAL (-12) , MARKER INITIAL (-25)
                , LHS_ELEMENT_EXPRESSION INITIAL (-16)
                , PART_OF INITIAL (-17) , OL2_COMPARE INITIAL (-18)
                , SCALAR_STRING_MARKER INITIAL (-50)
                , SCALAR_OP INITIAL (-15)
                , MAX_TEMPS_USED(0:8) INITIAL((9)(0)) , DOUBLE_SEQ
                INITIAL(28) , ILLEGAL_OP_IN_CODER INITIAL(10) , OES
                INITIAL(-7) , BOOLEAN INITIAL(6) ,UNDEFINED_TYPECODE
                INITIAL(30) , LHS_EE INITIAL(-11) , COMPARE1
                INITIAL(-20) , COMPARE2 INITIAL(-21) ,
                MATRIX_TIMES_SCALAR INITIAL(-19) , OP_SCALAR_MULTIPLY
                INITIAL(-10) , FREE INITIAL(-9) , SCALAR_MATRIX
                INITIAL(40)
             ) FIXED BINARY (15,0) STATIC                             ,
```

```
        OL2_COMPARATOR CHAR(2) VARYING STATIC      ,
    (
         ACT_20_INDEX ,
        SCALAR_EXP_POINTER , SAVER_POINTER , STK_PTR
    ) FIXED BINARY (31,0) STATIC                              ,
    (
        ELEMENT_EXPRESSION_FOUND
    ) BIT(1) STATIC                                           ,
    (
        TYPE_OF_ID
    ) FIXED BINARY (15,0) CONTROLLED                          ,
    (
        SCALAR_STRING_OPS ( -5 : -1 ) CHAR (2) VARYING INITIAL
        ( '**' , '*' , '/' , '-' , '+' )
    ) STATIC                                                  ,
    DCL ACTION_12_ROUTINE(-20:-1) LABEL STATIC ;
    DCL NESTED_COMPARE BIT(1) STATIC ;
    DCL ( BOOL_OP INITIAL(-19) , NOT_OP INITIAL(-20) )
     FIXED BIN (15,0) STATIC ;
    DCL (XPTR1,XPTR2) POINTER STATIC ;

DCL  DIGIT_STRINGS (0:25) CHAR(2) VARYING INITIAL
    ('0' , '1' , '2' ,
    '3' , '4' , '5' , '6' , '7' , '8' , '9' , '10' , '11' ,
    '12' , '13' , '14' , '15' , '16' , '17' , '18' , '19' ,
    '20' , '21' , '22' , '23' , '24' , '25' )                 ,
   YES INITIAL ('1'B) , NO INITIAL ('0'B),
   ,(B_STRING , L_STRING) CHAR(200) VARYING ;
     DCL TRANSPOSE_ELMT CHAR(4) STATIC ;
    DCL UNIMINUS_FOUND BIT(1) STATIC INITIAL('0'B) ;
    DCL SEQ_ELEMENT CHAR(40) VARYING STATIC ;
    DCL SEMANTIC_ERROR BIT(1) STATIC ;
    DCL U82 CHAR(1) STATIC INITIAL(' ' /* 082 */ ) ;
```

```
/***************************************************************************/
/*                                                                         */
/*      ACTION ROUTINE 8 SCANS A PROSPECTIVE STATEMENT TO                  */
/*      DETERMINE IF IT IS AN ASSIGNMENT STATEMENT. IF AN ASSIGN-          */
/*      MENT STATEMENT IS NOT FOUND, THE PARSER VARIABLE OK                */
/*      IS SET TO ZERO TO INDICATE THE FAILURE. IF AN ASSIGN-              */
/*      MENT STATEMENT IS FOUND APPROPRIATE INITIALIZATION IS DONE         */
/*      BEFORE THE STATEMENT BEGINS TO BE PROCESSED.                       */
/*                                                                         */
/*         ERROR 8 INDICATES THAT UNMATCHED PARENTHESES EXIST              */
/*         IN THE STATEMENT BEING SCANNED.                                 */
/*                                                                         */
/***************************************************************************/

 ACTION_8:  /* INITIALIZE TO PROCESS AN ASSIGNMENT STATEMENT               */
  TEST_ASSIGN: DO WHILE( CHAR(INP) = ' ' ) ;
                 INP = INP+1 ;
             END ;
             A_STRING = CHAR(INP) ;
             IF A_STRING = ',' | A_STRING = '=' | A_STRING = '.' |
              A_STRING ='-' | A_STRING = '|' & CHAR(INP+1) = '_' |
              A_STRING = '<' THEN DO ;
                 INP = BEGINNING_OF_STATEMENT_PTR ;
                 STK_PTR = 0 ;
                 EXPRESSION_AREA = EMPTY ;
                 SEMANTIC_ERROR = NO ;
             /* SET COMMON SUB EXPRESSION POINTER */
                 CURRENT_TABLE_PTR = 0 ;
                 GO TO RETURN_TO_PARSER ;
             END ;
             IF A_STRING ¬= '(' THEN GO TO SOUT ;
             INP = INP + 1 ;
             PARN_COUNT = 1 ;
 SCAN_AGAIN: CALL SCAN_UNTIL_PASS( '(' , ')' , ';' ) ;
             A_STRING = CHAR(INP) ;
             INP = INP + 1 ;
             IF A_STRING = ')' THEN DO ;
                 PARN_COUNT = PARN_COUNT - 1 ;
                 IF PARN_COUNT = 0 THEN GO TO TEST_ASSIGN ;
                 GO TO SCAN_AGAIN ;
             END ;
             IF A_STRING = '(' THEN DO ;
                 PARN_COUNT = PARN_COUNT + 1 ;
                 GO TO SCAN_AGAIN ;
             END ;
             CALL #ERROR(8) ;
 SOUT:       INP = BEGINNING_OF_STATEMENT_PTR ;
             GO TO SET_OK_ZERO_AND_RETURN ;
```

```
/*************************************************************************/
/*                                                                       */
/*      ACTION ROUTINE 9 SETS A POINTER TO THE BEGINNING OF THE         */
/*      CURRENT STATEMENT BEING PROCESSED.                               */
/*                                                                       */
/*************************************************************************/

 ACTION_9:  /* SAVE POINTER TO BEGINNING OF STATEMENT                    */
            BEGINNING_OF_STATEMENT_PTR = INP ;
            GO TO RETURN_TO_PARSER ;


/*************************************************************************/
/*                                                                       */
/*      THE FOLLOWING ACTION ROUTINES ARE CALLED BY THE SYNTAX          */
/*      ANALYZER TO INDICATE THE OPERATIONAL SYMBOL CURRENTLY           */
/*      BEING PROCESSED. THESE ROUTINES ARE CALLED JUST BEFORE
/*      AN UNSTACK OPERATION IS TO BE DONE.                              */
/*                                                                       */
/*************************************************************************/

 ACTION_10: CURRENT_OP = PLUS;
            GO TO RETURN_TO_PARSER;
 ACTION_11: CURRENT_OP = MINUS;
            GO TO RETURN_TO_PARSER;
```

```
/************************************************************************************/
/*                                                                                  */
/*       ACTION ROUTINE 12 IS THE SECTION OF THE EXPRESSION                         */
/*       MODULE THAT DOES AN UNSTACK OPERATION ON THE TOP ENTRY(S)                  */
/*       IN THE PARSING STACK. A TRANSFER ON THE CURRENT OPERATOR                   */
/*       TYPE TO BE HANDLED IS MADE TO THE APPROPRIATE SUBSECTION                   */
/*       OF THE ROUTINE. IN THE SUBSECTION, AN OPERATOR NODE IS                     */
/*       BUILT AND THE STACK ENTRY OR ENTRIES ARE LINKED TO IT.                     */
/*       THE STACK ENTRY(FOR A UNARY OPERATOR) OR ENTRIES ( FOR                     */
/*       A BINARY OPERATOR) ARE THEN POPPED FROM THE STACK. ERROR                   */
/*       CHECKING IS DONE TO ASSUME THAT THE OPERANDS TO BE UN-                     */
/*       STACKED ARE COMPATIBLE AS TO DIMENSIONALITY AND TYPE.                      */
/*       THE NODE FIELDS OF THE OPERATOR RESULT NODE ARE FILLED                     */
/*       IN, AND A STACK ENTRY IS PUSHED FOR THE SUBTREE THAT                       */
/*       HAS JUST BEEN BUILT.                                                       */
/*       SUBTREE TYPE CODES USED SRE AS FOLLOWS:                                    */
/*          0 = SCALAR, 1 = FUNCTION , 2 = COLUMN VECTOR ,                          */
/*          3 = ROW VECTOR , 4 = MATRIX , 5 = NULL OPERAND ,                        */
/*          6 = BOOLEAN RESULT.                                                     */
/*                                                                                  */
/*          ERROR 6 INDICATES AN ATTEMPT TO ADD OPERANDS OF                         */
/*          DIFFERENT TYPES.                                                        */
/*                                                                                  */
/*          ERROR 9 INDICATES ILLEGAL OPERAND TYPES IN A SCALAR                     */
/*          STRING CONCATENATION.                                                   */
/*                                                                                  */
/*          ERROR 12 INDICATES AN ATTEMPT TO ADD OPERANDS                           */
/*          OF DIFFERING DIMENSIONALITY.                                            */
/*                                                                                  */
/*          ERROR 13 INDICATES AN ATTEMPT TO DIVIDE AN ARRAY                        */
/*          VARIABLE BY A NON SCALAR OPERAND.                                       */
/*                                                                                  */
/*          ERROR 14 INDICATES THAT A NON DECLARED OPERAND IN                       */
/*          AN INNER PRODUCT CONSTRUCT HAS NOT BEEN SPECIFIED                       */
/*          AS A PART OF ANOTHER ARRAY.                                             */
/*                                                                                  */
/*          ERROR 15 INDICATES THAT AN ILLEGAL OPERAND HAS BEEN                     */
/*          FOUND IN AN ARRAY**SCALAR CONSTRUCT.                                    */
/*                                                                                  */
/*          ERROR 16 INDICATES THAT AN ILLEGAL OPERAND, AS TO                       */
/*          DIMENSIONALITY, IS INVOLVED IN A MULTIPLY.                              */
/*                                                                                  */
/*          ERROR 17 INDICATES THAT AN ATTEMPT HAS BEEN MADE                        */
/*          TO APPLY THE NORM OPERATION TO A NON ARRAY OPERAND.                     */
/*                                                                                  */
/*          ERROR 18 INDICATES AN UNDER FLOW IN THE EXPRESSION STACK.               */
/*                                                                                  */
/*          ERROR 19 INDICATES THAT ILLEGAL OPERAND TYPES ARE                       */
/*          PRESENT IN A FUNCTION OPERATION.                                        */
/*                                                                                  */
/*          ERROR 20 INDICATES THAT THE OPERAND TYPES ARE ILLEGAL                   */
/*          IN AN ASSIGNMENT STATEMENT.                                             */
/*                                                                                  */
/*          ERROR 21 INDICATES THAT AN ATTEMPT HAS BEEN MADE                        */
/*          TO MAKE AN ASSIGNMENT TO AN IDENITY OPERATOR.                           */
/*                                                                                  */
/*          ERROR 55 INDICATES THAT ILLEGAL OPERANDS ARE PRESENT                    */
/*          IN A BOOLEAN OPERATION.                                                 */
/*                                                                                  */
/************************************************************************************/
```

```
ACTION_12:
          ALLOCATE TREE_NODE IN (EXPRESSION_AREA) ;
          $TYPE_CODE=CURRENT_OP;
          STRING_POINTER , SEQ_#_PTR = NULL ;
          RLINK,LLINK=NULL;
          NEGATE_TAG , TRANSPOSE_TAG , IDENTITY_TAG = NO ;
          #_OF_TIMES_TO_USE = 1 ;
          PART_SIZE = 0 ;

          CALL GOTO( ACTION_12_ROUTINE(CURRENT_OP) ) ;


          ACTION_12_ROUTINE_PLUS: ACTION_12_ROUTINE_MINUS:
          IF SUBTREE_TYPE(STK_PTR) = SCALAR & SUBTREE_TYPE(
           STK_PTR-1) = SCALAR THEN GO TO SCALAR_OP_FOUND ;
              IF SUBTREE_TYPE(STK_PTR)¬=SUBTREE_TYPE(STK_PTR-1)
               THEN CALL #ERROR(6) ;
              XPTR1=SUBTREE_PTR(STK_PTR);
              XPTR2=SUBTREE_PTR(STK_PTR-1);
              IF XPTR1->$#DIMENSIONS¬=XPTR2->$#DIMENSIONS THEN
               CALL #ERROR(12);
              $#DIMENSIONS=XPTR1->$#DIMENSIONS;
              TYPE_EXP=XPTR1->TYPE_EXP;
              GO TO LINK_BINARY_OP;


          ACTION_12_ROUTINE_DIVIDE:
          IF SUBTREE_TYPE(STK_PTR) = SCALAR & SUBTREE_TYPE(
           STK_PTR-1) = SCALAR THEN GO TO SCALAR_OP_FOUND ;
              IF SUBTREE_TYPE(STK_PTR)¬=0 THEN CALL #ERROR(13);
              XPTR1=SUBTREE_PTR(STK_PTR-1);
              $#DIMENSIONS=XPTR1->$#DIMENSIONS;
              TYPE_EXP=XPTR1->TYPE_EXP;
              GO TO LINK_BINARY_OP;


          ACTION_12_ROUTINE_INNER_PRODUCT:
          DO I = 0,1 ;
              IF SUBTREE_TYPE(STK_PTR-I) ¬= 2 THEN DO ;
                  XPTR1 = SUBTREE_PTR(STK_PTR-I) ;
                  IF XPTR1 -> $TYPE_CODE ¬= PART_OF THEN CALL
                   #ERROR(14) ;
                  END ;
              END ;
              SUBTREE_TYPE(STK_PTR-1)=0;
              $#DIMENSIONS=0;
              TYPE_EXP=SCALAR;
              GOTO LINK_BINARY_OP;
```

```
ACTION_12_ROUTINE_EXPONENTIATE:
IF SUBTREE_TYPE(STK_PTR) = SCALAR & SUBTREE_TYPE(
 STK_PTR-1) = SCALAR THEN GO TO SCALAR_OP_FOUND ;
    IF SUBTREE_TYPE(STK_PTR)¬=0 THEN CALL #ERROR(15);
    IF SUBTREE_TYPE(STK_PTR-1)¬=4 THEN CALL #ERROR(15);
    XPTR1=SUBTREE_PTR(STK_PTR-1);
    $#DIMENSIONS=XPTR1->$#DIMENSIONS;
    TYPE_EXP=MATRIX;
    GO TO LINK_BINARY_OP;


ACTION_12_ROUTINE_NORM:
    IF SUBTREE_TYPE(STK_PTR) < 2 THEN
     CALL #ERROR(17) ;
    SUBTREE_TYPE(STK_PTR)=0;
    $#DIMENSIONS=0;
    TYPE_EXP=SCALAR;
    GO TO LINK_UNIARY_OP ;


ACTION_12_ROUTINE_UNIMINUS:
    XPTR1=SUBTREE_PTR(STK_PTR);
    $#DIMENSIONS=XPTR1->$#DIMENSIONS;
    TYPE_EXP=XPTR1->TYPE_EXP;
    GO TO LINK_UNIARY_OP;


ACTION_12_ROUTINE_TRANSPOSE:
    IF SUBTREE_TYPE(STK_PTR)=2
     THEN SUBTREE_TYPE(STK_PTR),TYPE_EXP=3;
    ELSE IF SUBTREE_TYPE(STK_PTR)=3
     THEN     SUBTREE_TYPE(STK_PTR),TYPE_EXP=2;
/* IGNORE TRANSPOSE OF A SCALAR                                  */
    ELSE IF SUBTREE_TYPE(STK_PTR) <= 1 THEN DO;
        FREE TREE_NODE;
        GOTO RETURN_TO_PARSER;
        END;
    ELSE DO;
        TYPE_EXP=MATRIX;  END;
    XPTR1=SUBTREE_PTR(STK_PTR);
    $#DIMENSIONS=XPTR1->$#DIMENSIONS;
    GO TO LINK_UNIARY_OP;
```

```
ACTION_12_ROUTINE_MULTIPLY:
IF SUBTREE_TYPE(STK_PTR) = SCALAR & SUBTREE_TYPE(
 STK_PTR-1) = SCALAR THEN GO TO SCALAR_OP_FOUND ;
    IF SUBTREE_TYPE(STK_PTR)<=1 THEN DO;
        XPTR1=SUBTREE_PTR(STK_PTR-1);
        $#DIMENSIONS=XPTR1->$#DIMENSIONS;
        TYPE_EXP=XPTR1->TYPE_EXP;
        CURRENT_OP = MATRIX_TIMES_SCALAR ;
        GO TO LINK_BINARY_OP;
        END;

    IF SUBTREE_TYPE(STK_PTR-1)<=1 THEN DO;
        SUBTREE_TYPE(STK_PTR-1)=SUBTREE_TYPE(STK_PTR);
        XPTR1=SUBTREE_PTR(STK_PTR);
        $#DIMENSIONS=XPTR1->$#DIMENSIONS;
        TYPE_EXP=XPTR1->TYPE_EXP;
        CURRENT_OP = MATRIX_TIMES_SCALAR ;
        GO TO LINK_BINARY_OP;
        END;

    IF SUBTREE_TYPE(STK_PTR-1)=2 THEN DO;
        IF SUBTREE_TYPE(STK_PTR)=3 THEN DO;
            SUBTREE_TYPE(STK_PTR-1)=4;
            $#DIMENSIONS=2;
    TYPE_EXP=MATRIX;
            GO TO LINK_BINARY_OP;
            END;
        ELSE CALL #ERROR(16);
    GO TO RETURN_TO_PARSER ;
        END;
    IF SUBTREE_TYPE(STK_PTR-1)=3 THEN DO;
        IF SUBTREE_TYPE(STK_PTR)=2 THEN DO ;
            SUBTREE_TYPE(STK_PTR-1) = 0 ;
            $#DIMENSIONS = 0 ;
            CURRENT_OP = INNER_PRODUCT ;
            GO TO LINK_BINARY_OP ;
        END ;
        IF SUBTREE_TYPE(STK_PTR)=4 THEN DO;
            XPTR1=SUBTREE_PTR(STK_PTR);
            $#DIMENSIONS=XPTR1->$#DIMENSIONS-1;
        IF $#DIMENSIONS=1 THEN DO;
             SUBTREE_TYPE(STK_PTR-1)=3 ;
            TYPE_EXP=ROW_VEC ;  END;
        ELSE SUBTREE_TYPE(STK_PTR-1),TYPE_EXP=MATRIX;
            GO TO LINK_BINARY_OP;
            END;
        END;
    IF SUBTREE_TYPE(STK_PTR-1)=4 THEN DO;
        TYPE_EXP=MATRIX;
        XPTR1=SUBTREE_PTR(STK_PTR-1);
        XPTR2=SUBTREE_PTR(STK_PTR);
        $#DIMENSIONS=XPTR1->$#DIMENSIONS+
         XPTR2->$#DIMENSIONS-2;
        IF $#DIMENSIONS=1 THEN
         SUBTREE_TYPE(STK_PTR-1),TYPE_EXP=COL_VEC;
        GO TO LINK_BINARY_OP;
        END;
```

```
ACTION_12_ROUTINE_EQUAL:
    IF SUBTREE_TYPE(STK_PTR) = SUBTREE_TYPE(STK_PTR-1) |
     SUBTREE_TYPE(STK_PTR) = 5 & SUBTREE_TYPE(STK_PTR-1)
     > 1 | SUBTREE_TYPE(STK_PTR) = 0 & SUBTREE_TYPE
     (STK_PTR-1) >= 2 & SUBTREE_TYPE(STK_PTR-1) <= 4
      THEN DO ;
    XPTR1 = SUBTREE_PTR(STK_PTR-1) ;
    XPTR2 = SUBTREE_PTR(STK_PTR) ;
    IF XPTR1 -> IDENTITY_TAG THEN CALL #ERROR(21) ;
    $#DIMENSIONS=XPTR2->$#DIMENSIONS;
    TYPE_EXP=XPTR1->TYPE_EXP;
    GO TO LINK_BINARY_OP;
    END ;
    CALL #ERROR(20) ;
    GO TO RETURN_TO_PARSER;


ACTION_12_ROUTINE_LHS_E_E:
    STRING_POINTER = POINTER_TO_STRING( A_STRING ,
        EXPRESSION_AREA);
$#DIMENSIONS = 0 ;
GO TO LINK_UNIARY_OP ;


ACTION_12_ROUTINE_SEQUENCE:
    SEQ_#_PTR = POINTER_TO_STRING( TEMPSTRING ,
        EXPRESSION_AREA);
A12S: XPTR1 = SUBTREE_PTR(STK_PTR) ;
    $#DIMENSIONS = XPTR1 -> $#DIMENSIONS ;
    GO TO LINK_UNIARY_OP ;


ACTION_12_ROUTINE_PART_OF:
XPTR1=SUBTREE_PTR(STK_PTR);
    TREE_NODE=XPTR1->TREE_NODE;
    XPTR1,SUBTREE_PTR(STK_PTR)=NODE_POINTER;
XPTR1->STRING_POINTER=POINTER_TO_STRING( B_STRING,
                        EXPRESSION_AREA);
     GO TO CHECK_PRINT;


ACTION_12_ROUTINE_CONCATENATE:
    IF SUBTREE_TYPE(STK_PTR-1) ¬= 0 |
     SUBTREE_TYPE(STK_PTR) ¬= 0 THEN CALL #ERROR(9) ;
        $#DIMENSIONS=0;
    CURRENT_OP = SCALAR_OP ;
        GO TO LINK_BINARY_OP;
    GO TO RETURN_TO_PARSER;


ACTION_12_ROUTINE_SEPARATOR:
    $#DIMENSIONS=0;
  STRING_POINTER = ADDR(COMMA_DUMMY) ;
    CURRENT_OP = SCALAR_OP ;
    GO TO LINK_BINARY_OP;
```

```
ACTION_12_ROUTINE_FUNCTION:
    IF SUBTREE_TYPE(STK_PTR-1) = 1 THEN DO ;
        SUBTREE_TYPE(STK_PTR-1) = 0 ;
        $#DIMENSIONS=0;
    GO TO LINK_BINARY_OP;
        END;
    ELSE CALL #ERROR(19) ; /* ILLEGAL FUNCTION */
    GO TO RETURN_TO_PARSER;


ACTION_12_ROUTINE_MODIFY:
    $#DIMENSIONS=0;
    SUBTREE_TYPE(STK_PTR)=0;
    GO TO LINK_UNIARY_OP;


ACTION_12_ROUTINE_OL2_COMPARE:
/* PROCESS AN OL/2 COMPARE OPERATOR                                 */
IF SUBTREE_TYPE(STK_PTR) = SCALAR & SUBTREE_TYPE(
 STK_PTR-1) = SCALAR THEN CURRENT_OP = SCALAR_OP ;
 ELSE IF SUBTREE_TYPE(STK_PTR) = SCALAR |
 SUBTREE_TYPE(STK_PTR-1) = SCALAR THEN
 CURRENT_OP = COMPARE2 ;
 ELSE CURRENT_OP = COMPARE1 ;
STRING_POINTER = POINTER_TO_STRING( OL2_COMPARATOR ,
 EXPRESSION_AREA ) ;
 SUBTREE_TYPE(STK_PTR - 1) = BOOLEAN ;
$#DIMENSIONS = 0 ;
GO TO LINK_BINARY_OP ;


ACTION_12_ROUTINE_BOOL_OP:
    IF SUBTREE_TYPE(STK_PTR) ¬= BOOLEAN | SUBTREE_TYPE
     (STK_PTR-1) ¬= BOOLEAN THEN CALL #ERROR(55) ;
    $#DIMENSIONS = 0 ;
    STRING_POINTER = POINTER_TO_STRING(TEMPSTRING,
     EXPRESSION_AREA) ;
    CURRENT_OP = SCALAR_OP ;
    GO TO LINK_BINARY_OP ;


ACTION_12_ROUTINE_NOT_OP:
    IF SUBTREE_TYPE(STK_PTR) ¬= BOOLEAN THEN
     CALL #ERROR(55) ;
    $#DIMENSIONS = 0 ;
    GO TO LINK_UNIARY_OP ;
```

```
/**********************************************************************/
/*                                                                    */
/*      THIS SECTICN OF THE ROUTINE DOES THE REPEATED SUBEXPRESS-     */
/*      ION HANDLING AS DESCRIBED IN ALGORITHM RS OF THE TEXT.        */
/*                                                                    */
/**********************************************************************/

          LINK_UNIARY_OP:
          IF CURRENT_OP = SEQUENCE | CURRENT_OP = PART_OF |
           CURRENT_OP = LHS_ELEMENT_EXPRESSION THEN GO TO LINK_U ;
          DO K = 1 TO CURRENT_TABLE_PTR ;
               IF SUBTREE_TABLE(K).POINTER_TO_OP1 = SUBTREE_PTR(
                STK_PTR) & SUBTREE_TABLE(K).OPERATION =
                CURRENT_OP THEN DO ;
                    XPTR1 =  SUBTREE_TABLE(K).POINTER_TO_RESULT ;
                    XPTR1 -> #_OF_TIMES_TO_USE = XPTR1 ->
                     #_OF_TIMES_TO_USE + 1 ;
                    SUBTREE_PTR(STK_PTR) = XPTR1 ;
                    XPTR1 = XPTR1 -> RLINK ;
                    IF XPTR1 -> RLINK ¬= NULL THEN XPTR1 ->
                     #_OF_TIMES_TO_USE = XPTR1 ->
                     #_OF_TIMES_TO_USE - 1 ;
                    GO TO RETURN_TO_PARSER ;
               END ;
          END ;
          CURRENT_TABLE_PTR = CURRENT_TABLE_PTR + 1 ;
          SUBTREE_TABLE(CURRENT_TABLE_PTR).POINTER_TO_OP1 =
           SUBTREE_PTR(STK_PTR) ;
          SUBTREE_TABLE(CURRENT_TABLE_PTR).POINTER_TO_OP2 = NULL ;
          SUBTREE_TABLE(CURRENT_TABLE_PTR).OPERATION = CURRENT_OP ;
          SUBTREE_TABLE(CURRENT_TABLE_POINTER).POINTER_TO_RESULT =
           NODE_POINTER ;


/**********************************************************************/
/*                                                                    */
/*      THIS SECTION OF ACTION ROUTINE 12 DOES THE LINKING OF         */
/*      AN OPERAND NODE TO THE OPERATOR NODE FOR UNARY OPERATORS.     */
/*                                                                    */
/**********************************************************************/

          LINK_U:
          RLINK = SUBTREE_PTR(STK_PTR) ;
          CHECK_PRINT:
              SUBTREE_PTR(STK_PTR)=NODE_POINTER;
              IF STK_PTR = 0 THEN CALL #ERROR(18) ;
              GO TO RETURN_TO_PARSER;
 SCALAR_OP_FOUND:
          STRING_POINTER = POINTER_TO_STRING( SCALAR_STRING_OPS(
           CURRENT_OP) , EXPRESSION_AREA ) ;
          CURRENT_OP = SCALAR_OP ;
          $#DIMENSIONS = 0 ;
```

```
LINK_BINARY_OP:
IF CURRENT_OP = SCALAR_OP | CURRENT_OP = EQUAL |
 CURRENT_OP = COMPARE1 | CURRENT_OP = COMPARE2
 THEN GO TO LINK_B ;
DO K = 1 TO CURRENT_TABLE_PTR ;
     IF SUBTREE_TABLE(K).POINTER_TO_OP2 = SUBTREE_PTR(
      STK_PTR) & SUBTREE_TABLE(K).OPERATION =
     CURRENT_OP & SUBTREE_TABLE(K).POINTER_TO_OP1 =
      SUBTREE_PTR(STK_PTR-1) THEN DO ;
          XPTR1 =  SUBTREE_TABLE(K).POINTER_TO_RESULT ;
          XPTR1 -> #_OF_TIMES_TO_USE = XPTR1 ->
           #_OF_TIMES_TO_USE + 1 ;
          STK_PTR = STK_PTR - 1 ;
          SUBTREE_PTR(STK_PTR) = XPTR1 ;
          XPTR2 = XPTR1 -> LLINK ;
          XPTR1 = XPTR1 -> RLINK ;
          IF XPTR1 -> RLINK ¬= NULL THEN XPTR1 ->
           #_OF_TIMES_TO_USE = XPTR1 ->
           #_OF_TIMES_TO_USE - 1 ;
          IF XPTR2 -> RLINK ¬= NULL THEN XPTR2 ->
           #_OF_TIMES_TO_USE = XPTR2 ->
           #_OF_TIMES_TO_USE - 1 ;
          GO TO RETURN_TO_PARSER ;
     END ;
END ;
CURRENT_TABLE_PTR = CURRENT_TABLE_PTR + 1 ;
SUBTREE_TABLE(CURRENT_TABLE_PTR).POINTER_TO_OP1 =
 SUBTREE_PTR(STK_PTR-1) ;
SUBTREE_TABLE(CURRENT_TABLE_PTR).POINTER_TO_OP2 =
 SUBTREE_PTR(STK_PTR) ;
SUBTREE_TABLE(CURRENT_TABLE_PTR).OPERATION = CURRENT_OP ;

/*************************************************************************/
/*                                                                       */
/*      THIS SECTION OF ACTION ROUTINE 12 LINKS THE OPERAND NODES        */
/*      TO THE OPERATOR NODE FOR A BINARY OPERATION.                     */
/*                                                                       */
/*************************************************************************/

LINK_B :
    LLINK=SUBTREE_PTR(STK_PTR-1);
    RLINK=SUBTREE_PTR(STK_PTR);
    $TYPE_CODE = CURRENT_OP ;
    STK_PTR=STK_PTR-1;
    IF STK_PTR = 0 THEN CALL #ERROR(18) ;
    SUBTREE_PTR(STK_PTR)=NODE_POINTER;
    GO TO RETURN_TO_PARSER;
```

```
/**********************************************************************/
/*                                                                    */
/*      ACTION ROUTINE 13 STACKS A MARKER IN THE PARSING STACK TO    */
/*      INDICATE THE BEGINNING OF A POSSIBLE SERIES OF MULTIPLY      */
/*      OPERATIONS.                                                   */
/*                                                                    */
/**********************************************************************/

 ACTION_13: SUBTREE_TYPE(STK_PTR+1)=SUBTREE_TYPE(STK_PTR);
            SUBTREE_PTR(STK_PTR+1)=SUBTREE_PTR(STK_PTR);
            SUBTREE_TYPE(STK_PTR)=MARKER;
            STK_PTR=STK_PTR+1;
            GO TO RETURN_TO_PARSER;


/**********************************************************************/
/*                                                                    */
/*      ACTION 14 SETS THE CURRENT OPERATION TO BE INVOLVED IN       */
/*      AN UNSTACK TO BE THE MULTIPLY OPERATION.                     */
/*                                                                    */
/*      ACTION 15 SETS THE CURRENT OPERATION TO BE INVOLVED IN       */
/*      AN UNSTACK TO BE THE DIVIDE OPERATION.                       */
/*                                                                    */
/**********************************************************************/

 ACTION_14: CURRENT_OP=MULTIPLY;
            GO TO RETURN_TO_PARSER;

 ACTION_15: CURRENT_OP=DIVIDE;
            GO TO RETURN_TO_PARSER ;
```

```
/***************************************************************************/
/*                                                                         */
/*     ACTION ROUTINE 16 CONSULTS THE MULTIPLY PRECEDENCE                  */
/*     MATRIX TO DETERMINE IF AN UNSTACK CAN BE DONE. IF SO,               */
/*     ACTION ROUTINE 12 IS CALLED TO DO THE UNSTACK, AND THE              */
/*     ABOVE PROCEDURE IS REPEATED. OTHERWISE, IF AN UNSTACK CANNOT */
/*     BE DONE, THE ROUTINE IMMEDIATLY RETURNS TO THE PARSER.              */
/*                                                                         */
/***************************************************************************/

 ACTION_16: IF CURRENT_OP = CIVIDE THEN GO TO ACTION_12 ;
 AGAIN_16:  IF PRECEDENCE_TABLE(SUBTREE_TYPE(STK_PTR-1),
               SUBTREE_TYPE(STK_PTR))=0 THEN DO ;
                  IF SUBTREE_TYPE(STK_PTR-1)=3 &
                   SUBTREE_TYPE(STK_PTR) ¬= 2 &
                   SUBTREE_TYPE(STK_PTR-2)=0 THEN DO ;
                  STK_PTR=STK_PTR-1;
                  CURRENT_OP = MULTIPLY ;
                  CALL ACT(12);
                  SUBTREE_TYPE(STK_PTR+1)=SUBTREE_TYPE(STK_PTR+2);
                  SUBTREE_PTR(STK_PTR+1)=SUBTREE_PTR(STK_PTR+2);
                  STK_PTR=STK_PTR+1;
                  END;
            CURRENT_OP = MULTIPLY ;
            CALL ACT(12) ;
            IF SUBTREE_TYPE(STK_PTR-1)=MARKER THEN GO TO
             RETURN_TO_PARSER;
              GO TO AGAIN_16 ;
            END ;
            GO TO RETURN_TO_PARSER ;
```

```
/**************************************************************************/
/*                                                                        */
/*     ACTION ROUTINE 17 UNSTACKS ALL RESIDUAL ELEMENTS IN THE            */
/*     PARSING STACK, DOWN TO A MARKER, WHEN A SEQUENCE OF MULTIPLY       */
/*     OPERATIONS HAS ENDED. THE PRECEDENCE MATRIX IS CONSULTED TO        */
/*     DETERMINE IF ALL THE MULTIPLICATIONS ARE LEGAL.                    */
/*                                                                        */
/*     ERROR 11 INDICATES AN ILLEGAL MULTIPLICATION.                      */
/*                                                                        */
/**************************************************************************/

 ACTION_17: IF SUBTREE_TYPE(STK_PTR-1)=MARKER THEN DO;
               SUBTREE_TYPE(STK_PTR-1)=SUBTREE_TYPE(STK_PTR);
               SUBTREE_PTR(STK_PTR-1)=SUBTREE_PTR(STK_PTR);
               STK_PTR=STK_PTR-1;
               GO TO RETURN_TO_PARSER;
               END;
            IF PRECEDENCE_TABLE(SUBTREE_TYPE(STK_PTR-1),
             SUBTREE_TYPE(STK_PTR))¬=2 THEN DO;
               CURRENT_OP = MULTIPLY ;
               CALL ACT(12);
               GO TO ACTION_17;
               END;
               CALL #ERROR(11) ;
               GO TO RETURN_TO_PARSER ;

/**************************************************************************/
/*                                                                        */
/*     ACTION ROUTINES 18 AND 19 SET THE CURRENT OPERATOR TO BE           */
/*     CONSIDERED TO EXPONENTIATE OR UNARY MINUS REPPECTIVELY.            */
/*                                                                        */
/**************************************************************************/

 ACTION_18: CURRENT_OP=EXPONENTIATE;
            GO TO ACTION_12;

 ACTION_19: IF ¬ UNIMINUS_FOUND THEN GO TO RETURN_TO_PARSER ;
            CURRENT_OP=UNIMINUS ;
            UNIMINUS_FOUND = NO ;
            GO TO ACTION_12;
```

```
/****************************************************************************/
/*                                                                          */
/*      ACTION 20 DETERMINES IF A PARENTHESISED CONSTRUCT IS AN             */
/*      INNER PRODUCT BY COUNTING PARENTHESES AND COMMAS.                   */
/*                                                                          */
/****************************************************************************/

 ACTION_20: PARN_COUNT=1;
            ACT_20_INDEX=INP;
            ACT_20_COMMA_CHK:
                IF CHAR(ACT_20_INDEX)=',' & PARN_COUNT =1 THEN DO;
                    OK=0;
                    GO TO RETURN_TO_PARSER;
                    END;
                ELSE
                IF CHAR(ACT_20_INDEX)='(' THEN PARN_COUNT=PARN_COUNT+1;
            ELSE IF CHAR(ACT_20_INDEX)=')' THEN DO;
                    PARN_COUNT=PARN_COUNT-1;
                    IF PARN_COUNT=0 THEN DO;
                        OK=1;
                        GO TO RETURN_TO_PARSER;
                        END;
                    END;
                ACT_20_INDEX=ACT_20_INDEX+1;
                GO TO ACT_20_COMMA_CHK;

/****************************************************************************/
/*                                                                          */
/*      ACTION ROUTINES 21 AND 22 SET THE CURRENT OPERATION TO              */
/*      BE CONSIDERED, AND GO TO ACTION ROUTINE 12 TO DO AN                 */
/*      UNSTACK.                                                            */
/*                                                                          */
/****************************************************************************/

 ACTION_21: CURRENT_OP=TRANSPOSE;
            GO TO ACTION_12;

 ACTION_22: CURRENT_OP=SEQUENCE;
            GO TO ACTION_12;
```

```
/*****************************************************************************/
/*                                                                           */
/*      ACTION ROUTINE 23 STACKS AN IDENTIFER INTO THE EXPRESSION           */
/*      STACK, AFTER TESTING TO SEE IF IT IS OF THE PROPER                   */
/*      TYPE AND HAS BEEN DEFINED.                                           */
/*                                                                           */
/*         ERROR NOT_IMPLEMENTED INDICATES THE USE OF A FEATURE             */
/*         THAT IS NOT CURRENTLY IMPLEMENTED.                                */
/*                                                                           */
/*         ERROR 24 INDICATES THAT AN EXPRESSION STACK OVERFLOW             */
/*         HAS TAKEN PLACE .                                                 */
/*                                                                           */
/*****************************************************************************/

 ACTION_23: XPTR1=SEARCH(TEMPIDENT,J);
            IF XPTR1=NULL THEN DO;
                OK=0;
            GO TO RETURN_TO_PARSER;
                END;
            IF XPTR1 -> $TYPE_CODE = BLOCK_ARRAY | XPTR1 ->
             $TYPE_CODE = VECTOR_SPACE THEN CALL #ERROR(
             NOT_IMPLEMENTED ) ;
            IDENT_DEFINED = YES;
            SP = XPTR1 -> STRING_POINTER  ;
            B_STRING = STRINGS  ;
            SUBTREE_PTR(STK_PTR+1)=XPTR1;
            SUBTREE_TYPE(STK_PTR+1)=XPTR1->TYPE_EXP;
            STK_PTR = STK_PTR + 1 ;
            IF STK_PTR > MAX_STACK THEN CALL #ERROR(24) ;
            GO TO RETURN_TO_PARSER ;


/*****************************************************************************/
/*                                                                           */
/*      ACTION ROUTINE 24 SETS A POINTER TO THE BEGINNING OF A              */
/*      SCALAR STRING AND STACKS A MARKER IN THE EXPRESSION STACK.          */
/*                                                                           */
/*****************************************************************************/

 ACTION_24: /* SET POINTER TO BEGINNING OF A SCALAR STRING ,                */
            SCALAR_EXP_POINTER = INP ;
            STK_PTR = STK_PTR + 1 ;
            SUBTREE_TYPE(STK_PTR) = SCALAR_STRING_MARKER ;
            GO TO RETURN_TO_PARSER                                          ;
 ACTION_25: /* FINISH UP SCALAR PROCESSING                                  */
            IF SCALAR_EXP_POINTER ¬= INP THEN CALL
             BUILD_AND_STACK_SCALAR_NODE( SCALAR_EXP_POINTER , INP-1 );
            DO WHILE ( SUBTREE_TYPE( STK_PTR - 1 ) ¬=
             SCALAR_STRING_MARKER ) ;
                CURRENT_OP = CONCATENATE ;
                CALL ACT(12) ;
            END ;
            STK_PTR = STK_PTR - 1 ;
            SUBTREE_TYPE(STK_PTR) = SUBTREE_TYPE(STK_PTR+1) ;
            SUBTREE_PTR(STK_PTR) = SUBTREE_PTR(STK_PTR+1) ;
            GO TO RETURN_TO_PARSER                                          ;
```

```
/***************************************************************************/
/*                                                                         */
/*       ACTION ROUTINE 26 SAVES A POINTER TO THE BEGINNING OF             */
/*       AN IDENTIFIER.                                                    */
/*                                                                         */
/***************************************************************************/

 ACTION_26: /* SAVE A POINTER TO THE BEGINNING OF THE IDENTIFIER        */
            SAVER_POINTER = INP ;
            ELEMENT_EXPRESSION_FOUND = NO ;
            GO TO RETURN_TO_PARSER                                      ;


/***************************************************************************/
/*                                                                         */
/*       ACTION ROUTINE 27 PROCESSES A PARENTHESIZED CONSTRUCT             */
/*       AFTER AN IDENTIFIER, AND STACKS A NODE FOR THE IDENTIFIER,        */
/*       THE ACTION 70 ENTRY INTO ACTION 27 PROCESSES A PARENTHE-          */
/*       SIZED CONSTRUCT AND DOES NOT STACK A NODE.                        */
/*                                                                         */
/*          ERROR 11 INDICATES AN ILLEGAL MULTIPLICATION.
/*                                                                         */
/*          ERROR UNMATCHED_PARNS INDICATES UNMATCHED PARENTHESES.         */
/*                                                                         */
/***************************************************************************/

 ACTION_27: /* PROCESS PARENTHESIZED CONSTRUCT AFTER THE IDENTIFIER     */
            IF TYPE_OF_ID ¬= OL2_ID THEN GO TO BNOT                     ;
                IF SCALAR_EXP_POINTER ¬= SAVER_POINTER THEN
                 CALL BUILD_AND_STACK_SCALAR_NODE(
                 SCALAR_EXP_POINTER , SAVER_POINTER - 1 ) ;
                 SEQ_ELEMENT = TEMPSTRING ;
            IF IO_STAT THEN CALL CRUNCH2;
 ACTION_70:      A_STRING = ''                                          ;
                 PARN_COUNT = 1                                         ;
                 DO WHILE ( PARN_COUNT > 0 )                            ;
                      CALL SCAN_UNTIL_KEEP(')','(',';',',');
                      A_STRING = A_STRING || TEMPSTRING                 ;
                      CHARACTER_SCANNED = CHAR(INP)                     ;
                      IF CHARACTER_SCANNED = ')' THEN DO                ;
                           PARN_COUNT = PARN_COUNT-1                    ;
                           IF PARN_COUNT = 0 THEN GO TO BREADY          ;
                      END                                               ;
                      ELSE IF CHARACTER_SCANNED = '(' THEN
                      PARN_COUNT = PARN_COUNT + 1                       ;
                      ELSE IF CHARACTER_SCANNED = ';' THEN CALL
                       #ERROR( UNMATCHED_PARNS )                        ;
            ELSE IF CHARACTER_SCANNED = ',' THEN DO;
                  A_STRING=A_STRING||',';
            END;
            INP = INP + 1;
                 END                                                    ;
```

```
        BREADY: IF WHICH_ACTION_NUM = 70 THEN GO TO
         RETURN_TO_PARSER                                        ;
        IF IO_STAT THEN CALL CRUNCH;
        INP = INP + 1 ;
        SAVER_POINTER = INP  ;
        SCALAR_EXP_POINTER=INP;
        XPTR1 = SEARCH(TEMPIDENT,I) ;
        XPTR1 = XPTR1 -> STRING_POINTER ;
        TEMPIDENT = XPTR1->STRINGS ;
    CALL BUILD_AND_STACK_STRING_NODE('@OLERET(''0''B,' ||
    TRANSPOSE_ELMT||','||SEQ_ELEMENT||',' ||TEMPIDENT||
    ',' || A_STRING || ')' );
        NAMES_USED(-13) = YES ;
        ELEMENT_EXPRESSION_FOUND = YES ;
        GO TO SET_OK_ZERO_AND_RETURN ;
    BNOT:
        IF SCALAR_EXP_POINTER ¬= SAVER_POINTER THEN CALL
         BUILD_AND_STACK_SCALAR_NODE( SCALAR_EXP_POINTER ,
         SAVER_POINTER - 1 ) ;
        CALL BUILD_AND_STACK_SCALAR_NODE( SAVER_POINTER ,
         INP - 1 ) ;
        SUBTREE_TYPE ( STK_PTR ) = FUNCTION  ;
    GO TO RETURN_TO_PARSER                                       ;

/*************************************************************************/
/*                                                                       */
/*      ACTION ROUTINE 28 HANDLES A PL/1 FUNCTION THAT HAS AN OL/2       */
/*      ARGUMENT.                                                        */
/*                                                                       */
/*************************************************************************/

 ACTION_28: /* SEE IF PL1 FUNCTION WITH OL2 ARGUMENT */
    IF SUBTREE_TYPE( STK_PTR ) ¬= SCALAR THEN DO ;
         IF TYPE_OF_ID = OTHER THEN DO ;
              CURRENT_OP = MODIFY_TO_TYPE_PL1 ;
              GO TO ACTION_12 ;
         END ;
    END ;
    GO TO RETURN_TO_PARSER                                       ;
```

```
/*************************************************************************/
/*                                                                       */
/*      ACTION ROUTINES 29 AND 30 SET THE CURRENT OPERATOR TO            */
/*      BE CONSIDERED, FOR VARIOUS OL/2 OPERATIONS, AND TRANSFER TO      */
/*      ACTION ROUTINE 12, THE UNSTACKING ROUTINE.                       */
/*                                                                       */
/*************************************************************************/

 ACTION_29: /* BUILD AN ARGUEMANT SUBTREE                                */
           CURRENT_OP= SEPARATOR ;
           GO TO ACTION_12 ;

 ACTION_30: /* FINISH OFF A FUNCTION SUBTREE                             */
           CURRENT_OP = FUNCTION_TYPE ;
           SCALAR_EXP_POINTER = INP ;
           GO TO ACTION_12 ;


/*************************************************************************/
/*                                                                       */
/*      ACTION ROUTINE 31 PROSSES AN ELEMENT EXPRESSION CON-             */
/*      STRUCT , IF PRESENT, AFTER AN OL/2 IDENTIFIER.                   */
/*                                                                       */
/*************************************************************************/

 ACTION_31: /* HAS AN OL/2 IDENTIFIER ALONE BEEN FOUND ?                 */
           IF TYPE_OF_ID=OL2_ID & ELEMENT_EXPRESSION_FOUND THEN DO;
               INP = SAVER_POINTER  ;
               GO TO RETURN_TO_PARSER;
           END  ;
           IF TYPE_OF_ID = OL2_ID & ¬ ELEMENT_EXPRESSION_FOUND
            THEN DO ;
               IF SAVE_IDENT -> $#DIMENSIONS = 0 THEN DO  ;
                   IF SCALAR_EXP_POINTER ¬= SAVER_POINTER &
                    CHAR(SCALAR_EXP_POINTER) ¬= '-' THEN
                   CALL BUILD_AND_STACK_SCALAR_NODE (
                     SCALAR_EXP_POINTER , SAVER_POINTER - 1 )  ;
                   SP = SAVE_IDENT -> STRING_POINTER  ;
                   A_STRING = STRINGS  ;
                   CALL BUILD_AND_STACK_STRING_NODE (
                 '@OLSCAL(' || A_STRING || ',' ||TEMPSTRING ||')' );
                     NAMES_USED(-12) = YES ;
                   SCALAR_EXP_POINTER = INP  ;
                   OK = 1  ;
               END  ;
               ELSE  DO  ;
                   OK = 0  ;
                   FREE TYPE_OF_ID  ;
               END  ;
           END ;
           GO TO RETURN_TO_PARSER                                        ;
```

```
/**********************************************************************/
/*                                                                    */
/*      ACTION ROUTINES 32 AND 33 ARE INVOLVED WITH THE              */
/*      PROCESSING OF AN ARITHMETIC CONSTANT.                        */
/*                                                                    */
/**********************************************************************/

 ACTION_32: /* INITIALIZE TO TEST FOR AN ARITHMETIC CONSTANT         */
            TEMPCONST = '' ;
            GO TO RETURN_TO_PARSER                                   ;

 ACTION_33: /* HAS ONLY A '.' BEEN FOUND ?                           */
            IF TEMPCONST = '' THEN OK = 0 ;
            GO TO RETURN_TO_PARSER                                   ;


/**********************************************************************/
/*                                                                    */
/*      ACTION ROUTINES 34 AND 36 SET THE CURRENT OPERATION TO       */
/*      BE CONSIDERED FOR THE NORM AND INNER PRODUCT OPERATIONS      */
/*      AND TRANSFER TO THE UNSTACKING ROUTINES.                     */
/*                                                                    */
/**********************************************************************/

 ACTION_34: /* PROCESS A NORM                                        */
            CURRENT_OP = NORM ;
            SCALAR_EXP_POINTER = INP ;
            GO TO ACTION_12 ;

  ACTION_36: CURRENT_OP=INNER_PRODUCT;
            SCALAR_EXP_POINTER = INP ;
            GO TO ACTION_12 ;
```

```
/*************************************************************************/
/*                                                                       */
/*      ACTION ROUTINES 38 AND 39 ARE INVOLVED WITH IDENTIFIER          */
/*      PROCESSING: ACTION ROUTINE 39 DETERMINES THE TYPE OF            */
/*      IDENTIFIER PRESENT.                                             */
/*                                                                       */
/*************************************************************************/

 ACTION_38: /* DON'T NEED THIS ANY MORE                                  */
          FREE TYPE_OF_ID ;
          GO TO RETURN_TO_PARSER                                         ;

 ACTION_39: /* FIND THE TYPE OF THE IDENTIFIER                           */
          TEMPSTRING = '0' ;
          ALLOCATE TYPE_OF_ID ;
          TEMP_POINTER1 = SEARCH( TEMPIDENT , I ) ;
          IF TEMP_POINTER1 = NULL THEN DO ;
               INPUT_TYPE = IDENTIFIER_TYPECODE(I);
              IF IS_AN_OL2_ENTRY( TEMPIDENT ) THEN
               TYPE_OF_ID = OL2_ENTRY ;
              ELSE TYPE_OF_ID = OTHER ;
          END ;
          ELSE DO  ;
              TYPE_OF_ID = OL2_ID  ;
              SAVE_IDENT = TEMP_POINTER1  ;
          END  ;
          GO TO RETURN_TO_PARSER                                         ;


/*************************************************************************/
/*                                                                       */
/*      ACTION ROUTINES 35, 37, AND 40 ARE INVOLVED WITH SCALAR         */
/*      STRING PROCESSING.                                              */
/*                                                                       */
/*************************************************************************/

 ACTION_35: /* MOVE OVER A ',' OR A ')'                                  */
          SCALAR_EXP_POINTER = INP+1 ;
          GO TO RETURN_TO_PARSER;

 ACTION_37: /* PROCESS A POSSIBLE PREVIOUS SCALAR STRING                 */
          K = INP - 2 ;
 TO_37:   IF SCALAR_EXP_POINTER ¬= K THEN DO ;
              CALL BUILD_AND_STACK_SCALAR_NODE(
               SCALAR_EXP_POINTER , K-1) ;
          END ;
          GO TO RETURN_TO_PARSER                                         ;

 ACTION_40: /* UNSTACK SCALAR MARKER                                     */
          STK_PTR = STK_PTR - 1 ;
          GO TO SET_OK_ZERO_AND_RETURN ;
```

```
/***************************************************************************/
/*                                                                         */
/*      ACTION ROUTINE 41 CALLS THE EXPRESSION TREE CODER , IF NO          */
/*      ERRORS HAVE BEEN FOUND, WHEN THE EXPRESSION PARSING                */
/*      ALGORITHM HAS FINISHED BUILDING THE EXPRESSION TREE.               */
/*                                                                         */
/*         ERROR 23 INDICATES THAT A SEMANTIC ERROR WAS                    */
/*      OF THE EXPRESSION TREE SHOULD NOT BE DONE.                         */
/*         FOUND BY THE EXPRESSION PARSER, AND THAT CODING                 */
/*         OF THE EXPRESSION TREE SHOULD NOT BE DONE.                      */
/*                                                                         */
/***************************************************************************/

 ACTION_41: /* CALL EXPRESSION TREE CODER */
           IF SEMANTIC_ERROR THEN CALL #ERROR(23) ; ELSE
           CALL CODER_#_1 ( SUBTREE_PTR(STK_PTR) ) ;
           GO TO RETURN_TO_PARSER;


/***************************************************************************/
/*                                                                         */
/*      ACTION ROUTINES 42 AND 43 FINISH UP PROCESSING OF AN               */
/*      ASSIGNMENT STATEMENT, WHICH MAY HAVE A MULTIPLE LEFT               */
/*      HAND SIDE, BY CALLING ACTION 12, THE UNSTACKING ROUTINE,           */
/*      WITH THE CURRENT OPERATION TO BE CONSIDERED SET TO                 */
/*      EQUAL.                                                             */
/*                                                                         */
/*         ERROR 23 INDICATES THAT A SEMANTIC ERROR WAS                    */
/*         FOUND BY THE EXPRESSION PARSER, AND THAT CODING                 */
/*         OF THE EXPRESSION TREE SHOULD NOT BE DONE.                      */
/*                                                                         */
/*         ERROR 30 INDICATES THAT AN ILLEGAL OL/2 OPERATION               */
/*         HAS BEEN FOUND.                                                 */
/*                                                                         */
/***************************************************************************/

 ACTION_42: /* HANDLE ASSIGNMENT , MULTIPLE IF NECESSARY                   */
           DO I = STK_PTR TO 2 BY -1 ;
               CURRENT_OP = EQUAL ;
               CALL ACT(12) ;
           END ;
           GO TO RETURN_TO_PARSER ;

 ACTION_43: /* HANDLE ERRORS IN AN ASSIGNMENT STATEMENT */
           IF SEMANTIC_ERROR THEN CALL #ERROR(23) ;
           ELSE DO ;
               IF FARTHEST > 6 THEN INP = FARTHEST ;
               CALL #ERROR(30) ;
           END ;
           CALL SCAN_UNTIL_PASS(';' , ' ' ) ; /* SECOND IS 082 */
               INP = INP + 1 ;
           GO TO RETURN_TO_PARSER ;
```

```
/*************************************************************************/
/*                                                                       */
/*      ACTION ROUTINES 44 AND 45 SET THE CURRENT OPERATION              */
/*      TO BE CONSIDERED AND TRANSFER TO THE UNSTACKING ROU-             */
/*      TINE, ACTICN 12.                                                 */
/*                                                                       */
/*************************************************************************/

 ACTION_44: /* PRCCESS PART-CF CONSTRUCT                                 */
           CURRENT_OP = PART_OF ;
           GO TO ACTION_12 ;

 ACTION_45: /* PROCESS OL/2 ELEMENT EXPRESSION ON LHS OF                 */
           /*  AN ASSIGNMENT OPERATOR                                    */
           CURRENT_OP = LHS_ELEMENT_EXPRESSION ;
           GO TO ACTION_12 ;


/*************************************************************************/
/*                                                                       */
/*      ACTION ROUTINE 46 STACKS AN OL/2 NULL OPERAND.                   */
/*************************************************************************/

 ACTION_46: /* PROCESS A "NULL" OPERAND                                  */
           STK_PTR = STK_PTR +1 ;
           SUBTREE_PTR(STK_PTR) = $OL2NULL ;
           SUBTREE_TYPE(STK_PTR) = 5 ;
           GO TO RETURN_TO_PARSER ;
```

```
/*****************************************************************************/
/*                                                                           */
/*      ACTION ROUTINE 104 PROCESSES AN OL/2 SEQUENCE EXPRESSION.           */
/*                                                                           */
/*          ERROR 53 INDICATES THAT UNMATCHED SEQUENCE BRACKETS              */
/*          EXIST.                                                           */
/*                                                                           */
/*****************************************************************************/

 ACTION_104:/* PICK UP SEQUENCE EXPRESSION                                  */
           IF IO_STAT THEN CALL CRUNCH2;
           CALL SCAN_UNTIL_KEEP( '_|' , ';' , ' ' ) ;
           /* LAST IS 082 */
           IF CHAR(INP) = ';' THEN CALL #ERROR (211)  ;
           /* TEST FOR 082 */
           IF CHAR(INP) = ' ' THEN DO ;
                CALL #ERROR(53) ;
                GO TO SET_OK_ZERO_AND_RETURN ;
           END ;
           IF IO_STAT THEN CALL CRUNCH;
           GO TO RETURN_TO_PARSER ;


/*****************************************************************************/
/*                                                                           */
/*      ACTION ROUTINE 113 PROCESSES SUBARRAY INDICES WITHIN A               */
/*      PART_OF CONSTRUCT.                                                   */
/*                                                                           */
/*****************************************************************************/

 ACTION_113:/*   PROCESS SUBARRAY INDICES                                   */
           TEST_FOR_PART_OF:
           I = INP ;
           CALL SCAN_UNTIL_PASS( ' ' /* 082 */, '<' , '>' ,
            '=' , '|' , '&' , ';' , '¬' ) ;
           IF CHAR(INP) ¬= '>' THEN DO ;
                IF CHAR(INP) = ' ' /* 082 */ THEN CALL #ERROR(55) ;
                GO TO SET_OK_ZERO_AND_RETURN ;
           END ;
```

```
J = INP ;
INP = INP + 1 ;
DO WHILE(CHAR(INP) = ' ' ) ;
     INP = INP + 1 ;
END ;
DCL CH CHAR (1) STATIC;
CH = CHAR(INP);
IF CH ¬= '''' THEN IF CH ¬= '*' THEN
IF CH ¬= '/' THEN IF CH ¬= '+' THEN
IF CH ¬= '-' THEN IF CH ¬= '=' THEN
IF CH ¬= ';' THEN IF CH ¬= ')' THEN
IF CH ¬= '<' THEN IF ¬IO_STAT THEN
IF ¬I_STAT THEN
      GO TO SET_OK_ZERO_AND_RETURN;
ELSE DO;   IF CH ¬= ',' THEN
  GO TO SET_OK_ZERO_AND_RETURN;                END;
ELSE IF CH ¬= ',' THEN IF CH ¬= '|' THEN IF CH ¬= ':'
      THEN GO TO SET_OK_ZERO_AND_RETURN;
IF IDENT_DEFINED = NO THEN DO ;
     INP = J ;
     GO TO RETURN_TO_PARSER ;
END ;
INP = I ;
IF IO_STAT THEN CALL CRUNCH2;


/*************************************************************************/
/*                                                                       */
/*      ACTION ROUTINES 155, 156, AND 157 SET A FLAG TO INDICATE         */
/*      THE PRESENCE OF AN OL/2 NEGATE OR TRANSPOSE OPERATOR.            */
/*                                                                       */
/*************************************************************************/

ACTION_155:
          UNIMINUS_FOUND = YES ;
          GO TO RETURN_TO_PARSER ;

ACTION_156:
          TRANSPOSE_ELMT = '''1''B' ;
          GO TO RETURN_TO_PARSER ;

ACTION_157:
          TRANSPOSE_ELMT = '''0''B' ;
          GO TO RETURN_TO_PARSER ;
```

```
/***************************************************************************/
/*                                                                         */
/*      ACTION ROUTINES 158 THROUGH 168 ARE INVOLVED WITH THE              */
/*      PROCESSING OF OL/2 BOOLEAN OPERATORS. TRANSFERS ARE MADE           */
/*      TO THE UNSTACKING ROUTINE WHEN NECESSARY TO BUILD SUBTREES.        */
/*                                                                         */
/*        ERRORS 53 AND 55 INDICATE UNMATCHED SYMBOLS IN THE               */
/*         INPUT STRING.                                                   */
/*                                                                         */
/***************************************************************************/

 ACTION_158:
          TEMPIDENT = B_STRING ;
          GO TO RETURN_TO_PARSER ;

 ACTION_159:/* PROCESS A BOOLEAR "OR" */
          TEMPSTRING = '|' ;
          CURRENT_OP = BOOL_OP ;
          GO TO ACTION_12 ;

 ACTION_160:/* PROCESS A BOOLEAN "AND" */
          TEMPSTRING = '&' ;
          CURRENT_OP = BOOL_OP ;
          GO TO ACTION_12 ;

 ACTION_161:/* SCAN FOR A BOOLEAN EXPRESSION , RATHER THAN AN ARITH-
          METIC EXPRESSION */
          NESTED_COMPARE = NO ;
          K = INP ;
          PARN_COUNT = 1 ;
  ISCAN:  CALL SCAN_UNTIL_PASS( '|' , '&' , ';' , '(' , ')' ,
           '¬' , '>' , '<' , '=' ) ;
          A_STRING = CHAR(INP) ;
          IF A_STRING = '|' | A_STRING = '&' THEN
  IOUT:   DO ;
              INP = K ;
              GO TO SET_OK_ZERO_AND_RETURN ;
          END ;
          IF A_STRING = ';' THEN DO;
              CALL #ERROR(53) ;
              GO TO IOUT ;
          END ;
```

```
          IF A_STRING = '(' THEN DO ;
               PARN_COUNT = PARN_COUNT + 1 ;
               INP = INP + 1 ;
               GO TO ISCAN ;
          END ;
          IF A_STRING = ')' THEN DO ;
               PARN_COUNT = PARN_COUNT - 1 ;
               IF PARN_COUNT = 0 THEN DO ;
                    IF ¬ NESTED_COMPARE THEN GO TO IOUT ;
                    INP = K ;
                    GO TO RETURN_TO_PARSER ;
               END ;
               INP = INP + 1 ;
               GO TO ISCAN ;
          END ;
          IF A_STRING = ' ' /* 082 */ THEN DO ;
               CALL #ERROR(55) ;
               GO TO SET_OK_ZERO_AND_RETURN ;
          END ;
          NESTED_COMPARE = YES ;
               INP = INP + 1 ;
          GO TO ISCAN ;

ACTION_162:/* HANDLE A "¬" OPERATOR */
          CURRENT_OP = NOT_OP ;
          GO TO ACTION_12 ;

  ACTION_163:
          STK_PTR = 0 ;
          EXPRESSION_AREA = EMPTY ;
          SEMANTIC_ERROR = NO ;
          CURRENT_TABLE_PTR = 0 ;
          GO TO RETURN_TO_PARSER ;

  ACTION_164:
          CURRENT_OP = OL2_COMPARE ;
          GO TO ACTION_12 ;
```

```
ACTION_165:
          OL2_CCMPARATOR = '' ;
          K = INP ;
          DO WHILE ( CHAR(INP) = ' ' ) ;
              INP = INP - 1 ;
          END ;
          CALL MOVCHAR(OL2_COMPARATOR,INP-1,INP) ;
          INP = K ;
          GO TO RETURN_TO_PARSER ;

ACTION_166:
          OL2_CCMPARATOR = CHAR(INP-1) ;
          GO TO RETURN_TO_PARSER ;

ACTION_167:
          SUBTREE_TYPE(STK_PTR) = BOOLEAN ;
          GO TO RETURN_TO_PARSER ;

ACTION_168:
          IF SEMANTIC_ERROR THEN CALL #ERROR(23) ;
          ELSE CALL CODER_#_1(SUBTREE_PTR(STK_PTR)) ;
          XPTR1 = SUBTREE_PTR(STK_PTR) ;
          XPTR1 = XPTR1->STRING_POINTER ;
          OUTPUT_BUFFER = XPTR1->STRINGS ;
          OUTPUT_BUFFER = 'IF ' || OUTPUT_BUFFER || ' THEN ' ;
          CALL SKIP_AND_OUTPUT ;
          OUTPUT_BUFFER = 'DO /* THEN DO */ ;' ;
          CALL SKIP_AND_OUTPUT ;
          GO TO RETURN_TO_PARSER ;
```

```
/************************************************************************/
/*                                                                      */
/*      THE FOLLOWING PROCEDURES ARE INTERNAL TO THE SEMANTIC          */
/*      ROUTINES OF THE EXPRESSOON HANDLING MODULE.                    */
/*                                                                      */
/************************************************************************/


/************************************************************************/
/*                                                                      */
/*      THE PROCEDURE SKIP_AND_OUTPUT WRITES THE PL/1 REPRE-           */
/*      SENTATION OF THE OL/2 SOURCE CODE FROM THE STRING              */
/*      VARIABLE OUTPUT_BUFFER, WHICH THE COMPILER HAS FILLED,         */
/*      INTO A FILE THAT WILL LATER BE PROCESSED BE THE PL/1           */
/*      COMPILER.                                                      */
/*                                                                      */
/************************************************************************/

 SKIP_AND_OUTPUT: PROCEDURE                                            ;
           DCL DIVIDE BUILTIN  ;
           DCL ( I , J , K ) FIXED BIN (15,0)  STATIC ;
           I = DIVIDE ( LENGTH ( OUTPUT_BUFFER ) , 71 , 15 , 0 )  ;
           J = MOD ( LENGTH ( OUTPUT_BUFFER ) , 71 )  ;
           DO K = 1 TO I  ;
               PUT FILE(SYSPUNCH) EDIT (SUBSTR ( OUTPUT_BUFFER ,
                   71*K-70 , 71 ) ) ( X(1) , A(79) ) ;
               PUT FILE(PL1CODE)  EDIT (SUBSTR ( OUTPUT_BUFFER ,
                   71*K-70 , 71 ) ) ( X(1) , A(79) ) ;
           END  ;
           IF J ¬= 0 THEN DO ;
           PUT FILE(SYSPUNCH) EDIT (SUBSTR ( OUTPUT_BUFFER ,
                   71*I+1  , J  ) ) ( X(1) , A(79) )  ;
           PUT FILE(PL1CODE)  EDIT (SUBSTR ( OUTPUT_BUFFER ,
                   71*I+1  , J  ) ) ( X(1) , A(79) )  ;
           END ;
           OUTPUT_BUFFER = ''                                          ;
 END SKIP_AND_OUTPUT                                                   ;
```

```
/*************************************************************************/
/*                                                                       */
/*       THE PROCEDURE #ERROR HANDLES CL/2 ERRORS THAT HAVE              */
/*       BEEN FOUND.                                                     */
/*                                                                       */
/*************************************************************************/

 #ERROR: PROCEDURE ( ERROR_CODE_# )                                      ;
            DECLARE ERROR_CODE_# FIXED BINARY (15,0)                     ;
        IF ERROR_CODE_# >= 200 THEN ERROR_CODE_# = ERROR_CODE_# - 170 ;
        ELSE IF ERROR_CODE_# >= 8 & ERROR_CODE_# <= 21 | ERROR_CODE_#
             = 100 THEN SEMANTIC_ERROR = YES ;
        ERROR_PTR = ERROR_PTR + 1 ;
        ERROR_CODES(ERROR_PTR) = ERROR_CODE_# ;
        ERROR_ON_CARD(ERROR_PTR) = CARDNUM ;
        ERROR_NEAR_COLUMN(ERROR_PTR) = CARDCOL ;
        ERROR_NEAR_INP(ERROR_PTR) = INP ;
        IF ERROR_PTR = 50 THEN SIGNAL ERROR ;
 END #ERROR                                                              ;


/*************************************************************************/
/*                                                                       */
/*       THE PROCEDURE POINTER_TO_STRING ALLOCATES SPACE FOR             */
/*       A CHARACTER STRING THAT HAS BEEN PASSED TO IT, IN               */
/*       AN AREA RESVERVED FOR SUCH STRINGS, AND RETURNS A               */
/*       POINTER TO THE STRING.                                          */
/*                                                                       */
/*************************************************************************/

 POINTER_TO_STRING: PROCEDURE ( STRING , AREA ) RETURNS ( POINTER )     ;
            DECLARE STRING CHARACTER (*)    VARYING ,
                    AREA AREA(*);
            STRING_LENGTH = LENGTH ( STRING )                            ;
            ALLOCATE VARIABLE_STRING IN ( AREA )                         ;
            STRINGS = STRING                                             ;
            RETURN ( SP )                                                ;
 END POINTER_TO_STRING                                                   ;
```

```
/*************************************************************************/
/*                                                                       */
/*     THE FOLLOWING TWC PRCCEDURES BUILD A NODE FOR A SCALAR            */
/*     STRING AND A CHARACTER STRING RESPECTIVELY, AND STACK             */
/*     THE NODE IN THE EXPRESSION STACK.                                 */
/*                                                                       */
/*        ERROR 24 INDICATES AN EXPRESSICN STACK OVERFLOW.               */
/*************************************************************************/

BUILD_AND_STACK_SCALAR_NODE: PROCEDURE ( INP1 , INP2 )                  ;
   DCL ( INP1 , INP2 ) FIXED BINARY (31,0)                              ;
          A_STRING = ''                                                 ;
          CALL MUVCHAR( A_STRING , INP1 , INP2 )                        ;
          GO TO LAB1                                                    ;
BUILD_AND_STACK_STRING_NODE: ENTRY ( STRING )                           ;
   DCL STRING CHAR (100) VARYING                                        ;
          A_STRING = STRING                                             ;
          LAB1: STK_PTR = STK_PTR + 1                                   ;
          IF STK_PTR > MAX_STACK THEN CALL #ERROR(24) ;
          ALLOCATE TREE_NODE IN ( EXPRESSION_AREA )                     ;
          SUBTREE_PTR( STK_PTR ) = NODE_POINTER                         ;
          SUBTREE_TYPE( STK_PTR ) = SCALAR                              ;
          LLINK , RLINK = NULL                                          ;
          $#DIMENSIONS = 0                                              ;
          STRING_POINTER = POINTER_TO_STRING( A_STRING ,
           EXPRESSION_AREA )                                            ;
          #_OF_TIMES_TO_USE = UNDEFINED                                 ;
          $TYPE_CODE,TYPE_EXP=SCALAR;
          SEQ_#_PTR = NULL                                              ;
          TRANSPOSE_TAG , NEGATE_TAG , IDENTITY_TAG = NO                ;
          PART_SIZE = 0 ;
END BUILD_AND_STACK_SCALAR_NODE                                         ;
```

```
/*************************************************************************/
/*                                                                       */
/*       THE PROCEDURE SEARCH SEARCHES THE OL/2 SYMBOL TABLE FOR A       */
/*       SPECIFIC IDENTIFIER AND RETURNS A POINTER TO THE ROOT NODE      */
/*       ASSOCIATED WITH THE VARIABLE REPRESENTED BY THE IDENTIFIER.     */
/*                                                                       */
/*************************************************************************/

 SEARCH: PROCEDURE ( IDENTIFIER , J ) RETURNS ( POINTER )               ;
            DECLARE IDENTIFIER CHAR(32) VARYING , (TEMP_PP POINTER ,
             ( J , I ) FIXED BINARY (31,0) ) STATIC                      ;
            DO I = CURRENT_ID - 1 TO 1 BY -1                             ;
                 TEMP_PP = IDENTIFIER_NAME_POINTER(I)                    ;
                 IF TEMP_PP -> STRINGS = IDENTIFIER THEN DO              ;
                      J = I                                              ;
                      RETURN ( IDENTIFIER_NODE_POINTER(I) )              ;
                 END                                                     ;
            END                                                          ;
            J = 0                                                        ;
            RETURN ( NULL )                                              ;
 END SEARCH                                                              ;


 ACTION_54:
 SET_OK_ZERO_AND_RETURN: OK = 0                                          ;

 RETURN_TO_PARSER: RETURN                                                ;

  END ACT;
```

```
/************************************************************************/
/*                                                                      */
/*          OL/2 EXPRESSICN TREE CODER                                  */
/*                                                                      */
/*      THE CODER WORKS BASICALLY AS OUTLINED IN THE TEXT,              */
/*     SEE ALGORITHM CA. THE PROCEDURE IS CALLED FROM THE               */
/*      SEMANTIC ROUTINES WITH THE PROCEDURE ARGUMENT POINTING          */
/*      TO THE ROOT CF THE COMPLETED EXPRESSION TREE.                   */
/*                                                                      */
/************************************************************************/

   CODER_#_1:
        PROCEDURE ( ROOT_NODE_POINTER ) ;

/************************************************************************/
/*                                                                      */
/*      ERROR HANDLING FACILIES FOR INTERRUPTS THAT OCCUR DURING        */
/*      CODING.                                                         */
/*                                                                      */
/************************************************************************/

   ON ERROR BEGIN ;
        PUT SKIP LIST(CODE_STRING ) ;
        GO TO END_CODER ;
   END ;

/************************************************************************/
/*                                                                      */
/*      LOCAL DECLARATIONS FOR THE CODER.                               */
/*                                                                      */
/************************************************************************/

        DCL EXTEMP STATIC POINTER ;
        DCL ( ROOT_NODE_POINTER , (P , TEMP , XPTR1 , XPTR2 ) STATIC )
             POINTER ,
             1 STACK CONTROLLED ,
                  2 STK_P POINTER ,
                  2 STK_D BIT(1) ;
        DCL ( ( I , J ) STATIC , TEMPS_USED (0:8)  INITIAL ((9)(0)) )
             FIXED BIN (15,0) ;
        DCL FREE_TEMPS CHAR(9) VARYING CONTROLLED ,
             CODE_STRING (1:16) CHAR(100) VARYING STATIC ;
        DCL PREC_NEEDED BIT(1) STATIC ;
        DCL COPY_UP ENTRY ;
         DCL(  PTR POINTER , # FIXED BIN (15,0) ) STATIC ;
```

```
/**************************************************************************/
/*                                                                        */
/*       TREE TRAVERSAL ALGORITHM T FROM KNUTH.                           */
/*                                                                        */
/**************************************************************************/

        END_T1: P = ROOT_NODE_POINTER ;

        END_T2: IF P = NULL THEN GO TO END_T4 ;

        END_T3: ALLOCATE STACK ;
             STK_P = P ;
             STK_D = NO ;
             IF P->$TYPE_CODE = FUNCTION_TYPE THEN DO ;
                  ALLOCATE FREE_TEMPS ;
                  FREE_TEMPS = 'MARKER' ;
                   END ;
             P = P->RLINK ;
             GO TO END_T2 ;

        END_T4: IF ¬ ALLOCATION(STACK) THEN GO TO END_CODER ;
             IF ¬ STK_D THEN DO ;
                  STK_D = YES ;
                  P = STK_P->LLINK ;
                  GO TO END_T2 ;
             END ;
             P = STK_P ;
             FREE STACK ;

/**************************************************************************/
/*                                                                        */
/*       VISIT THE NEXT NODE IN A RIGHT ENDORDER SEQUENCE.                */
/*                                                                        */
/**************************************************************************/

        VISIT: TYPE_CODE = P->$TYPE_CODE ;
             IF TYPE_CODE ¬< 0 THEN GO TO END_T4 ;
```

```
        CODE_STRING = '' ;

/*****************************************************************************/
/*                                                                           */
/*        TRANSFER TO PROPER SECTION OF CODER BASED ON OPERATOR TYPE.        */
/*                                                                           */
/*****************************************************************************/



/*****************************************************************************/
/*                                                                           */
/*        THIS SECTION PROCESSES THE OL/2 OPERATORS UNARY MINUS              */
/*        AND MODIFY_TO_TYPE_PL1.                                            */
/*                                                                           */
/*****************************************************************************/

     OPERATOR_FOUND: IF TYPE_CODE <= UNIMINUS & TYPE_CODE >=
          MODIFY_TO_TYPE_PL1 THEN DO ;
               XPTR1 = P->RLINK ;
               P->STRING_POINTER = XPTR1->STRING_POINTER ;
               P->#_OF_TIMES_TO_USE = XPTR1->#_OF_TIMES_TO_USE ;
               P->NEGATE_TAG = XPTR1->NEGATE_TAG ;
               P->TRANSPOSE_TAG = XPTR1->TRANSPOSE_TAG ;
               IF TYPE_CODE = UNIMINUS THEN P->NEGATE_TAG = ¬
                (P->NEGATE_TAG) ;
               ELSE IF TYPE_CODE = TRANSPOSE THEN P->
                TRANSPOSE_TAG = ¬(P->TRANSPOSE_TAG) ;
               ELSE IF TYPE_CODE = MODIFY_TO_TYPE_PL1 THEN DO ;
                    XPTR2 = XPTR1->STRING_POINTER ;
                    A_STRING = XPTR2->STRINGS ;
                    P->STRING_POINTER = POINTER_TO_STRING(
                     A_STRING || ' -> #ROOT_NODE.$ORIGIN' ,
                     EXPRESSION_AREA) ;
               END ;
```

```
/***********************************************************************/
/*                                                                     */
/*      THIS SECTICN PROCESSES THE OL/2 OPERATOR SEQUENCE.             */
/*                                                                     */
/***********************************************************************/

                  ELSE DO ; /* TYPE CODE = SEQUENCE */
                       XPTR2 = XPTR1->SEQ_#_PTR ;
                       IF XPTR2 ¬= NULL THEN CALL #ERROR(DOUBLE_SEQ) ;
                       GO TO CODER_1 ;
                  END ;
                  P->SEQ_#_PTR = XPTR1->SEQ_#_PTR ;
        CODER_1:       P->$TYPE_CODE = XPTR1->$TYPE_CODE ;
                       P -> IDENTITY_TAG = XPTR1->IDENTITY_TAG ;
                       P->RLINK,P->LLINK = NULL ;
                       GO TO END_T4 ;
                  END ;


/***********************************************************************/
/*                                                                     */
/*      THIS SECTION PROCESSES THE OL/2 OPERATOR FUNCTION_TYPE */
/*      AND ALSO HANDLES OPERATIONS INVOLVING STRICTLY SCALAR          */
/*      OPERANDS.                                                      */
/*                                                                     */
/***********************************************************************/

                  IF TYPE_CODE = SCALAR_OP | TYPE_CODE = FUNCTION_TYPE
                   THEN DO ;
                       PREC_NEEDED = NO ;
                       TEMP = P->LLINK ;
                       EXTEMP = P->RLINK ;
                       XPTR1 =  TEMP->STRING_POINTER ;
                       XPTR2 = EXTEMP->STRING_POINTER ;
                       A_STRING = XPTR1->STRINGS ;
            IF  TEMP -> NEGATE_TAG THEN A_STRING = '-(' ||
             A_STRING || ')' ;
                       TEMP = P-> STRING_POINTER ;
                       IF TEMP ¬= NULL THEN DO ;
                            B_STRING = TEMP-> STRINGS ;
                            IF B_STRING = ' , ' THEN DO ;
                                 # = 0 ;
                                 TEMP = P-> RLINK ;
```

```
CODER_2:                          IF TEMP->$TYPE_CODE >= 0 THEN DO;
                             TEMP->#_OF_TIMES_TO_USE = TEMP->
                              #_OF_TIMES_TO_USE - 1 ;
                             IF TEMP->#_OF_TIMES_TO_USE = 0
                              THEN DO ;
                                  ALLOCATE FREE_TEMPS ;
                                  PTR = TEMP->STRING_POINTER ;
                                  FREE_TEMPS = PTR->STRINGS ;
                             END ;
                             END ;
                        # = # + 1 ;
                        IF #=2 THEN GO TO CODER_3 ;
                        TEMP = P->LLINK ;
                        GO TO CODER_2 ;
CODER_3:                END ;
                        ELSE PREC_NEEDED = YES ;
                        A_STRING = A_STRING || B_STRING ;
                   END ;
                   B_STRING = XPTR2->STRINGS ;
    IF EXTEMP -> NEGATE_TAG THEN B_STRING = '-(' ||
     B_STRING || ')' ;
                   A_STRING = A_STRING || B_STRING ;


                   IF TYPE_CODE = FUNCTION_TYPE THEN DO ;
                        TEMPS_USED(0) = TEMPS_USED(0)+1 ;
                        B_STRING = '#TEMPO_' || DIGIT_STRINGS(
                         TEMPS_USED(0)) ;
                        OUTPUT_BUFFER = B_STRING || '=' ||
                         A_STRING || ');' ;
                        CALL SKIP_AND_OUTPUT ;
                        A_STRING = B_STRING ;
                        DO WHILE(FREE_TEMPS ¬= 'MARKER' ) ;
                             OUTPUT_BUFFER = 'CALL ' || NAMES
                              (FREE) || '(' || FREE_TEMPS || ');' ;
                             CALL SKIP_AND_OUTPUT ;
                             FREE FREE_TEMPS ;
                             NAMES_USED(FREE) = YES ;
                        END ;
                        FREE FREE_TEMPS ;
                   END ;
                   IF PREC_NEEDED THEN A_STRING = '(' ||
                    A_STRING || ')' ;
                   P->STRING_POINTER = POINTER_TO_STRING(A_STRING,
                    EXPRESSION_AREA) ;
                   P->RLINK,P->LLINK = NULL ;
                   P->$TYPE_CODE = SCALAR ;
                   GO TO END_T4 ;
              END ;
```

```
        IF TYPE_CODE = LHS_ELEMENT_EXPRESSION THEN GO TO END_T4;


/***************************************************************************/
/*                                                                         */
/*      THIS SECTION PROCESSES THE OL/2 OPERATORS PLUS, MINUS,             */
/*      AND MULTIPLY.                                                      */
/*                                                                         */
/***************************************************************************/

          IF TYPE_CODE = PLUS | TYPE_CODE = MINUS | TYPE_CODE =
           MULTIPLY THEN DO ;
               PTR = P->LLINK ;
               # = 2 ;
               CALL COPY_UP ;
               PTR = P->RLINK ;
               # = 7 ;
               CALL COPY_UP ;
               IF TYPE_CODE = MINUS THEN DO ;
                    IF CODE_STRING(8) = '''0''B' THEN CODE_STRING(8)
                     = '''1''B' ; ELSE CODE_STRING(8) = '''0''B' ;
                    TYPE_CODE = PLUS ;
               END ;
     CODE_STRING(12) = '''1''B' ;
     CODE_STRING(13) = '''0''B' ;
     CODE_STRING(14) = '''0''B' ;
     CODE_STRING(15) = '0' ;
     J = P->$#DIMENSIONS ;
     TEMPS_USED(J) = TEMPS_USED(J)+1 ;
     CODE_STRING(16) = '$TEMP' || DIGIT_STRINGS(J) || '_' ||
      DIGIT_STRINGS(TEMPS_USED(J)) ;
     IF TEMPS_USED(J) > MAX_TEMPS_USED(J) THEN MAX_TEMPS_USED(J) =
      TEMPS_USED(J) ;
               CODE_STRING(1) = NAMES(TYPE_CODE) ;
               NAMES_USED(TYPE_CODE) = YES ;
               NAMES_USED(PLUS) = YES ;
               P->$TYPE_CODE = UNDEFINED_TYPECODE ;
               P->RLINK,P->LLINK = NULL ;
               P->STRING_POINTER = POINTER_TO_STRING(CODE_STRING(16)
                , EXPRESSION_AREA ) ;
               GO TO PRINT_CODE ;
          END ;
```

```
/***************************************************************************/
/*                                                                         */
/*        THIS SECTION PROCESSES THE OL/2 OPERATOR EQUAL.                  */
/*                                                                         */
/***************************************************************************/

            IF TYPE_CODE = EQUAL THEN DO ;
                 XPTR1 = P->RLINK ;
                 XPTR2 = P-> LLINK ;
                 IF XPTR1->$#DIMENSIONS >0 THEN DO ;
                      PTR = XPTR1 ;
                      # = 2 ;
                      CALL COPY_UP ;
                      CODE_STRING(7) = '''1''B' ;
                      CODE_STRING(8) = '''0''B' ;
                      CODE_STRING(9) = '''0''B' ;
                      CODE_STRING(10) = '0' ;
                      CODE_STRING(11) = '$OLNULL' ;
                      PTR = XPTR2 ;
                      # = 12 ;
                      CALL COPY_UP ;
                      CODE_STRING(12) =='''0''B' ;
                      IF ALLOCATION(STACK) THEN DO;
                           P->$TYPE_CODE = XPTR1->$TYPE_CODE ;
                           P->NEGATE_TAG = XPTR1->NEGATE_TAG ;
                           P->TRANSPOSE_TAG = XPTR1 -> TRANSPOSE_TAG ;
                           P->SEQ_#_PTR = XPTR1->SEQ_#_PTR ;
                           P->RLINK,P->LLINK = NULL ;
                           P->STRING_POINTER = XPTR1->STRING_POINTER ;
                      END ;
                      CODE_STRING(1) = NAMES(PLUS) ;
                      NAMES_USED(PLUS) = YES ;
                      GO TO PRINT_CODE ;
                 END ;
```

```
IF XPTR2->$#DIMENSIONS>0 THEN DO ;
     CODE_STRING(1) = NAMES(NORM) ;
     NAMES_USED(NORM) = YES ;
     PTR = XPTR2 ;
     # = 2 ;
     CALL COPY_UP ;
     TEMP = XPTR1->STRING_POINTER ;
     CODE_STRING(7) = TEMP->STRINGS ;
     IF XPTR1->NEGATE_TAG THEN CODE_STRING(7)=
      '-(' || CODE_STRING(7) || ')' ;
     CODE_STRING(8) = '6' ;
     IF ALLOCATION(STACK) THEN DO ;
          P->NEGATE_TAG = XPTR1->NEGATE_TAG ;
          P->$TYPE_CODE = SCALAR ;
          P->STRING_POINTER = TEMP ;
     END ;
     P->RLINK,P->LLINK = NULL ;
     GO TO PRINT_CODE ;
END ;


IF XPTR2 -> $TYPE_CODE = LHS_ELEMENT_EXPRESSION THEN
DO;
  CODE_STRING(1) = '@OLEASS';
  NAMES_USED(-13) = YES;
  CODE_STRING(2) = '''0''B,''0''B'  ;
  PTR = XPTR2 -> RLINK;
  TEMP = PTR -> SEQ_#_PTR;
IF TEMP = NULL THEN CODE_STRING(3) = '0' ; ELSE
  CODE_STRING(3) = TEMP->STRINGS;
  TEMP=PTR -> STRING_POINTER;
  CODE_STRING(4) = TEMP->STRINGS;
  TEMP = XPTR1 -> STRING_POINTER;
  CODE_STRING(5) = TEMP->STRINGS;
  TEMP = XPTR2 -> STRING_POINTER;
  CODE_STRING(6) = TEMP -> STRINGS;
  GO TO COPY_UP_NODE;
END;
```

```
    IF XPTR2->$TYPE_CODE = SCALAR_MATRIX THEN DO ;
           CODE_STRING(1) = NAMES(LHS_EE) ;
           NAMES_USED(LHS_EE) = YES ;
           TEMP = XPTR2->STRING_POINTER ;
           CODE_STRING(2) = TEMP->STRINGS ;
            TEMP = XPTR2 -> SEQ_#_PTR;
            IF TEMP = NULL THEN CODE_STRING(3) = '0';
            ELSE CODE_STRING(3) = TEMP->STRINGS;
           TEMP = XPTR1 -> STRING_POINTER ;
            CODE_STRING(4) = TEMP->STRINGS;
            IF XPTR1->NEGATE_TAG THEN CODE_STRING(4)='-('
             || CODE_STRING(4) || ')' ;

COPY_UP_NODE:
           IF ALLOCATION(STACK) THEN DO ;
                P->STRING_POINTER = XPTR1->STRING_POINTER ;
                P->NEGATE_TAG = XPTR1->NEGATE_TAG ;
                P->RLINK,P->LLINK = NULL ;
                P->$TYPE_CODE = SCALAR ;
           END ;
           GO TO PRINT_CODE ;
      END ;

      /* BOTH SCALARS */
      TEMP = XPTR2->STRING_POINTER ;
      A_STRING = TEMP->STRINGS ;
      OUTPUT_BUFFER = A_STRING || ' = ' ;
      TEMP = XPTR1->STRING_POINTER ;
      CODE_STRING(1) = TEMP->STRINGS ;
      IF XPTR1->NEGATE_TAG THEN CODE_STRING(1) = '-(' ||
       CODE_STRING(1) || ')' ;
      OUTPUT_BUFFER = OUTPUT_BUFFER || CODE_STRING(1) ||
       ';' ;
      CALL SKIP_AND_OUTPUT ;
      IF ALLOCATION(STACK) THEN DO ;
           P->STRING_POINTER = XPTR1->STRING_POINTER ;
           P->NEGATE_TAG = XPTR1->NEGATE_TAG ;
           P->RLINK,P->LLINK = NULL ;
           P->$TYPE_CODE = SCALAR ;
      END ;
      GO TO END_T4 ;
 END ;
```

```
/***************************************************************************/
/*                                                                         */
/*      THIS SECTION PROCESSES THE OL/2 OPERATIONS MATRIX_                 */
/*     TIMES_SCALAR, DIVIDE, AND EXPONENTIATE._                            */
/*                                                                         */
/***************************************************************************/

          IF TYPE_CODE = MATRIX_TIMES_SCALAR | TYPE_CODE = DIVIDE |
           TYPE_CODE = EXPONENTIATE THEN DO ;
               IF TYPE_CODE = MATRIX_TIMES_SCALAR THEN DO ;
                    XPTR1 = P->RLINK ;
                    IF XPTR1->$#DIMENSIONS = 0 THEN DO ;
                         XPTR2 = XPTR1 ;
                         XPTR1 = P->LLINK ;
                    END ;
                    ELSE XPTR2 = P->LLINK ;
                         TYPE_CODE= OP_SCALAR_MULTIPLY ;
               END ;
               ELSE DO ;
                    XPTR1 = P->LLINK ;
                    XPTR2 = P->RLINK ;
               END ;
               CODE_STRING(1) = NAMES(TYPE_CODE) ;
               TEMP = XPTR2->STRING_POINTER ;
               CODE_STRING(2) = TEMP->STRINGS ;
               IF XPTR2->NEGATE_TAG THEN CODE_STRING(2) =
                     '-(' || CODE_STRING(2) || ')' ;
                     IF TYPE_CODE = DIVIDE THEN DO ;
                          CODE_STRING(2) = '1.00/(' || CODE_STRING(2)
                           || ')' ;
                          TYPE_CODE=OP_SCALAR_MULTIPLY ;
                     END ;
               NAMES_USED(TYPE_CODE) = YES ;
               PTR = XPTR1 ;
               # = 3 ;
               CALL COPY_UP ;
     CODE_STRING(8) = '''1''B' ;
     CODE_STRING(9) = '''0''B' ;
     CODE_STRING(10) = '''0''B' ;
     CODE_STRING(11) = '0' ;
     J = P->$#DIMENSIONS ;
     TEMPS_USED(J) = TEMPS_USED(J)+1 ;
     CODE_STRING(12) = '$TEMP' || DIGIT_STRINGS(J) || '_' ||
      DIGIT_STRINGS(TEMPS_USED(J)) ;
     IF TEMPS_USED(J) > MAX_TEMPS_USED(J) THEN MAX_TEMPS_USED(J) =
      TEMPS_USED(J) ;
               P->$TYPE_CODE = XPTR1->$TYPE_CODE ;
               P->RLINK,P->LLINK = NULL ;
               P->STRING_POINTER = POINTER_TO_STRING(CODE_STRING
                (12),EXPRESSION_AREA ) ;
               GO TO PRINT_CODE ;
          END ;
```

```
/*************************************************************************/
/*                                                                       */
/*       THIS SECTION PROCESSES THE OL/2 OPERATION INNER PROD.           */
/*                                                                       */
/*************************************************************************/

          IF TYPE_CODE = INNER_PRODUCT THEN DO ;
               CODE_STRING(1) = NAMES(CURRENT_INNER_PRODUCT) ;
               NAMES_USED(CURRENT_INNER_PRODUCT) = YES ;
               PTR = P->LLINK ;
               # = 2 ;
               CALL COPY_UP ;
               PTR = P->RLINK ;
               # = 7 ;
               CALL COPY_UP ;
               TEMPS_USED(0) = TEMPS_USED(0)+1 ;
               CODE_STRING(12) = '#TEMPO_' || DIGIT_STRINGS(
               TEMPS_USED(0)) ;
               P->STRING_POINTER = POINTER_TO_STRING(
                CODE_STRING(12) , EXPRESSION_AREA) ;
               P->$TYPE_CODE = SCALAR ;
               P->RLINK = NULL ;
               GO TO PRINT_CODE ;
     END ;

/*************************************************************************/
/*                                                                       */
/*       THIS SECTION PROCESSES THE OL/2 OPERATION NORM.                 */
/*                                                                       */
/*************************************************************************/

          IF TYPE_CODE = NORM THEN DO ;
               CODE_STRING(1)=NAMES(NORM) ;
               NAMES_USED(NORM) = YES ;
               PTR = P->RLINK ;
               # = 2 ;
               CALL COPY_UP ;
               TEMPS_USED(0) = TEMPS_USED(0)+1 ;
               CODE_STRING(7) = '#TEMPO_' || DIGIT_STRINGS(
               TEMPS_USED(0)) ;
               P->STRING_POINTER = POINTER_TO_STRING(
                CODE_STRING(7) , EXPRESSION_AREA) ;
               CODE_STRING(8) = DIGIT_STRINGS(CURRENT_NORM) ;
               P->$TYPE_CODE = SCALAR ;
               P->RLINK = NULL ;
               GO TO PRINT_CODE ;
          END ;
```

```
/*****************************************************************************/
/*                                                                           */
/*      THIS SECTION PROCESS THE OL/2 COMPARE OPERATIONS.                   */
/*                                                                           */
/*****************************************************************************/

        IF TYPE_CODE = COMPARE1 | TYPE_CODE = COMPARE2 THEN DO ;
            PTR = P->LLINK ;
            # = 1 ;
            CALL COPY_UP ;
            IF TYPE_CODE = COMPARE2 THEN DO ;
                XPTR1 = P->RLINK ;
                XPTR2 = XPTR1->STRING_POINTER ;
                CODE_STRING(6) = XPTR2->STRINGS ;
                IF XPTR1->NEGATE_TAG THEN CODE_STRING(6) =
                 '-(' || CODE_STRING(6) || ')' ;
            END ;
            ELSE DO ;
                PTR = P->RLINK ;
                # = 6 ;
                CALL COPY_UP ;
            END ;
            XPTR1 = P->STRING_POINTER ;
            A_STRING = XPTR1->STRINGS ;
  /* PUT IN NAMES USED  */
            IF TYPE_CODE = COMPARE1 THEN DO ;
                B_STRING = '@OLUCO' ;
                NAMES_USED(-14) = YES ;
            END ;
            ELSE DO ;
                B_STRING = '@OLOCS' ;
                NAMES_USED(-07) = YES ;
            END ;
            A_STRING = B_STRING || '(''' || A_STRING || '''' ;
            J = 1 ;
            DO WHILE (CODE_STRING(J) ¬= '') ;
                A_STRING = A_STRING || ',' || CODE_STRING(J) ;
                J = J+1 ;
            END ;
            A_STRING = A_STRING || ')' ;
            P->STRING_POINTER = POINTER_TO_STRING(A_STRING ,
             EXPRESSION_AREA ) ;
            P->RLINK,P->LLINK = NULL ;
            P->$TYPE_CODE = BOOLEAN ;
            GO TO END_T4 ;
        END ;
```

```
        IF TYPE_CODE = PART_OF THEN DO ;
            P -> RLINK = NULL ;
            GO TO END_T4 ;
/* PUT IN NAMES USED  */
        END ;
       IF TYPE_CODE = NOT_OP THEN DO ;
           XPTR1 = P-> RLINK ;
           XPTR2 = XPTR1 -> STRING_POINTER ;
           A_STRING = XPTR2 -> STRINGS ;
           P -> STRING_POINTER = POINTER_TO_STRING( '¬' ||
            A_STRING , EXPRESSION_AREA) ;
           P -> RLINK = NULL ;
           IF XPTR1 -> NEGATE_TAG | XPTR1->TRANSPOSE_TAG |
            XPTR1 -> SEQ_#_PTR ¬= NULL | XPTR1 -> IDENTITY_TAG
            THEN CALL #ERROR(55) ;
           GO TO END_T4 ;
        CALL #ERROR( ILLEGAL_OP_IN_CODER ) ;
        GO TO END_CODER ;


/***********************************************************************/
/*                                                                     */
/*       PRINT_CODE PRINTS THE PL/1 OBJECT CODE THAT HAS BEEN          */
/*       BUILT UP IN CODE_STRING.                                      */
/*                                                                     */
/***********************************************************************/

 PRINT_CODE: OUTPUT_BUFFER = 'CALL '  || CODE_STRING(1) || '(' ||
             CODE_STRING(2) ;
             DO J = 3 TO 16 WHILE(CODE_STRING(J) ¬= '' );
                  OUTPUT_BUFFER = OUTPUT_BUFFER || ' , ' || CODE_STRING
                   (J) ;
             END ;
             OUTPUT_BUFFER = OUTPUT_BUFFER || ');' ;
             CALL SKIP_AND_OUTPUT ;
             GO TO END_T4 ;
```

```
/**********************************************************************/
/*                                                                    */
/*      COPY_UP COPYS THE FIELDS IN THE NODE REFERENCED BY PTR       */
/*      INTO THE MEMBER OF CODE_STRING STARTING AT THE VALUE OF #.   */
/*                                                                    */
/**********************************************************************/

   COPY_UP: PROC ;
        DCL XPTR2 POINTER STATIC ;
        IF PTR->#_OF_TIMES_TO_USE > 0 THEN PTR->#_OF_TIMES_TO_USE
         = PTR->#_OF_TIMES_TO_USE - 1 ;
        IF PTR->#_OF_TIMES_TO_USE = 0 THEN CODE_STRING(#) = '''0''B' ;
         ELSE CODE_STRING(#) = '''1''B' ;
        IF PTR->NEGATE_TAG THEN CODE_STRING(#+1) = '''1''B' ; ELSE
         CODE_STRING(#+1) = '''0''B' ;
        IF PTR->TRANSPOSE_TAG THEN CODE_STRING(#+2) = '''1''B' ; ELSE
         CODE_STRING(#+2) = '''0''B' ;
        XPTR2 = PTR->SEQ_#_PTR ;
        IF XPTR2 = NULL THEN CODE_STRING(#+3) = '0' ; ELSE
         CODE_STRING(#+3) = XPTR2->STRINGS ;
        XPTR2 = PTR->STRING_POINTER ;
        CODE_STRING(#+4) = XPTR2->STRINGS ;
   END COPY_UP ;


   END_CODER: IF TEMPS_USED(0) > MAX_TEMPS_USED(0) THEN
              MAX_TEMPS_USED(0) = TEMPS_USED(0) ;
   END CODER_#_1 ;

/**********************************************************************/
/*                                                                    */
/*      VARIABLES DECLARED IN ACT THAT ARE USED BY THE CODER.        */
/*                                                                    */
/**********************************************************************/

 DCL NAMES_USED(-30:-1) BIT(1) STATIC INITIAL((30) (1) '0'B ) ,

        DCL NAMES (-11:-1) CHAR(7) VARYING INITIAL
             ( '@OLAASS' , '@OLSMO' , '@OLFREE' , '@OLNORM' ,
              '@OLXXX' , '@OLINP' , '@OLEXP' , '@OLOMO' , '@OLSMO' ,
            '@OLOPS' , '@OLOPS' ) STATIC ;
```

APPENDIX C

```
*****OL/2*****OL/2***** COMPILER *****OL/2*****OL/2*****                SEQUENCE NUMBER     PAGE  1

THESIS_PROGRAM1: MAIN PROCEDURE;

      LET A,B, AND C BE ARRAYS OF ORDER(N),
             AND X,Y, AND Z BE VECTORS OF ORDER(N),
             AND ALPHA,BETA, AND GAMMA BE SCALARS ;

             /**********************************************
              *                                            *
              *  THIS OL/2 PROGRAM CONTAINS ASSIGNMENT     *
              *   STATEMENTS THAT ARE TO BE CODED.         *
              *                                            *
              **********************************************/

          A = B + C ;


          Z = (X' * -A)' + B * Y ;


          X = !! GAMMA*A*Z !! * B * Y ;


          BETA = ( (A + C) * X , !! X * Y' !! * Z) + ALPHA ;


          X = X + Y + B*Z ;


END THESIS_PROGRAM1;


OL/2   MATH LANGUAGE PREPROCESSOR DIAGNOSTICS.


NO ERRORS DETECTED.
PL/1 COMPILATION POSSIBLE.

END OF DIAGNOSTICS.


 END OF COMPILATION.
ECOMPILE TIME:   3.44  SEC.
```

```
/*********************************/
/*                               */
/*   PL/1 CODE GENERATED FROM    */
/*      TEST_PROGRAM1            */
/*                               */
/*********************************/


                /* A = B + C ;  */

13              CALL @OLOPS('1'B , '0'B , '0'B , 0 , $1B1 , '1'B , '0'B , '0'B , 0 , $1
                C1 , '1'B , '0'B , '0'B , 0 , $TEMP2_1);

14              CALL @OLOPS('0'B , '0'B , '0'B , 0 , $TEMP2_1 , '1'B , '0'B , '0'B , 0
                , $OLNULL , '0'B , '0'B , '0'B , 0 , $1A1);


                /* Z = (X' * -A)' + B * Y ;  */

16              CALL @OLOMO('1'B , '0'B , '0'B , 0 , $1B1 , '1'B , '0'B , '0'B , 0 , $1
                Y1 , '1'B , '0'B , '0'B , 0 , $TEMP1_1);

17              CALL @OLOMO('1'B , '0'B , '1'B , 0 , $1X1 , '1'B , '1'B , '0'B , 0 , $1
                A1 , '1'B , '0'B , '0'B , 0 , $TEMP1_2);

18              CALL @OLOPS('0'B , '0'B , '1'B , 0 , $TEMP1_2 , '0'B , '0'B , '0'B , 0
                , $TEMP1_1 , '1'B , '0'B , '0'B , 0 , $TEMPT_3);

19              CALL @OLOPS('0'B , '0'B , '0'B , 0 , $TEMP1_3 , '1'B , '0'B , '0'B , 0
                , $OLNULL , '0'B , '0'B , '0'B , 0 , $1Z1);


                /* X = !! GAMMA*A*Z !! * B * Y ;  */

21              CALL @OLOMO('1'B , '0'B , '0'B , 0 , $1B1 , '1'B , '0'B , '0'B , 0 , $1
                Y1 , '1'B , '0'B , '0'B , 0 , $TEMP1_1);

22              CALL @OLOMO('1'B , '0'B , '0'B , 0 , $1A1 , '1'B , '0'B , '0'B , 0 , $1
                Z1 , '1'B , '0'B , '0'B , 0 , $TEMP1_2);

23              CALL @OLSMO(GAMMA , '0'B , '0'B , '0'B , 0 , $TEMP1_2 , '1'B , '0'B , '
                0'B , 0 , $TEMP1_3);

24              CALL @OLNORM('0'B , '0'B , '0'B , 0 , $TEMP1_3 , #TEMPO_1 , 2);

25              CALL @OLSMO(#TEMPO_1 , '0'B , '0'B , '0'B , 0 , $TEMP1_1 , '1'B , '0'B
                , '0'B , 0 , $TEMPT_4);

26              CALL @OLOPS('0'B , '0'B , '0'B , 0 , $TEMP1_4 , '1'B , '0'B , '0'B , 0
                , $OLNULL , '0'B , '0'B , '0'B , 0 , $1X1);
```

```
/* BETA = ( (A + C) * X , !! X * Y' !! * Z) + ALPHA ;  */

28   CALL @OLONO('1'B , '0'B , '0'B , 0 , $1X1 , '1'B , '0'B , '1'B , 0 , $1
     Y1 , '1'B , '0'B , '0'B , 0 , $TEMP2_1);

29   CALL @OLNONN('0'B , '0'B , '0'B , 0 , $TEMP2_1 , #TEMP0_1 , 2);

30   CALL @OLSHO(#TEMP0_1 , '1'B , '0'B , '0'B , 0 , $1Z1 , '1'B , '0'B , '0
     'B , 0 , $TEMP1_1);

31   CALL @OLOPS('1'B , '0'B , '0'B , 0 , $1A1 , '1'B , '0'B , '0'B , 0 , $1
     C1 , '1'B , '0'B , '0'B , 0 , $TEMP2_2);

32   CALL @OLONO('0'B , '0'B , '0'B , 0 , $TEMP2_2 , '1'B , '0'B , '0'B , 0
     , $1X1 , '1'B , '0'B , '0'B , 0 , $TEMP1_2);

33   CALL @OLINP('0'B , '0'B , '0'B , 0 , $TEMP1_2 , '0'B , '0'B , '0'B , 0
     , $TEMP1_1 , #TEMP0_2);

34   BETA  = (#TEMP0_2+ALPHA );


/* X = X + Y + B*Z ;  */

36   CALL @OLONO('1'B , '0'B , '0'B , 0 , $1B1 , '1'B , '0'B , '0'B , 0 , $1
     Z1 , '1'B , '0'B , '0'B , 0 , $TEMP1_1);

37   CALL @OLOPS('1'B , '0'B , '0'B , 0 , $1X1 , '1'B , '0'B , '0'B , 0 , $1
     Y1 , '1'B , '0'B , '0'B , 0 , $TEMP1_2);

38   CALL @OLOPS('0'B , '0'B , '0'B , 0 , $TEMP1_2 , '0'B , '0'B , '0'B , 0
     , $TEMP1_1 , '1'B , '0'B , '0'B , 0 , $TEMPT_3);

39   CALL @OLOPS('0'B , '0'B , '0'B , 0 , $TEMP1_3 , '1'B , '0'B , '0'B , 0
     , $OLNULL , '0'B , '0'B , '0'B , 0 , $1X1);
```

```
*****OL/2*****OL/2***** COMPILER *****OL/2*****OL/2*****          SEQUENCE NUMBER     PAGE  1

THESIS_PROGRAM2: MAIN PROCEDURE;

      LET A,B, AND C BE ARRAYS OF ORDER(N),
            AND X,Y, AND Z BE VECTORS OF ORDER(N),
            AND ALPHA,BETA, AND GAMMA BE SCALARS ;

            /*******************************************
             *                                         *
             *  EACH OF THE ASSIGNMENT STATEMENTS IN   *
             *   THIS OL/2 PROGRAM CONTAINS OL/2 ERRORS. *
             *                                         *
             *******************************************/

         A = B / C ;


         Z' = X' * A + B * Y ;


         X = A' ;


         BETA = (X' * (A + C) , !! Y' * X !! * Z) + ALPHA ;


         X = X + Y + Z'*B ;

END THESIS_PROGRAM2;
```

```
OL/2   MATH LANGUAGE PREPROCESSOR DIAGNOSTICS.
SEVERE ERRORS.
OL/2 ERROR #13    SEQUENCE  14   NEAR COLUMN 22  NEAR " / C ;      "  ILLEGAL OL/2 DIVISION - ONLY DIVISION BY A SCALAR IS
   ALLOWED
OL/2 ERROR #23    SEQUENCE  14   NEAR COLUMN 22  NEAR " / C ;      "  ILLEGAL ARITHMETIC OPERATION IMPLIED IN STATEMENT

OL/2 ERROR #57    SEQUENCE  17   NEAR COLUMN 12  NEAR "      Z' = X"  STATEMENT NOT RECOGNIZED AS OL/2 OR PL/1

OL/2 ERROR #20    SEQUENCE  20   NEAR COLUMN 19  NEAR "= A' ;      "  CONFLICTING ATTRIBUTES OF LEFT AND RIGHT SIDES IN
 ASSIGNMENT STAT.
OL/2 ERROR #23    SEQUENCE  20   NEAR COLUMN 19  NEAR "= A' ;      "  ILLEGAL ARITHMETIC OPERATION IMPLIED IN STATEMENT

OL/2 ERROR #17    SEQUENCE  23   NEAR COLUMN 47  NEAR " X 11 * Z) "  ILLEGAL ARITHMETIC OPERATION IMPLIED IN STATEMENT

OL/2 ERROR #14    SEQUENCE  23   NEAR COLUMN 52  NEAR " * Z) + ALP"  INNER PRODUCT OF A TYPE OTHER THAN COLUMN VECTOR

OL/2 ERROR #23    SEQUENCE  23   NEAR COLUMN 61  NEAR "LPHA ;      "  ILLEGAL ARITHMETIC OPERATION IMPLIED IN STATEMENT

OL/2 ERROR # 6    SEQUENCE  26   NEAR COLUMN 29  NEAR "Z'*B ;      "  INCOMPATABLE DIMENSIONALITIES IN '+' OR '-'

PL/1 COMPILATION NOT POSSIBLE.

END OF DIAGNOSTICS.


 END OF COMPILATION.
ECOMPILE TIME:   2.53  SEC.
```

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-72-550	2.	3. Recipient's Accession No.

. Title and Subtitle

AN APPROACH TO THE COMPILATION OF ARRAY EXPRESSIONS IN THE OL/2 LANGUAGE

5. Report Date: September 1972

6.

. Author(s)

Dale Rade Jurich

8. Performing Organization Rept. No. UIUCDCS-R-72-550

. Performing Organization Name and Address

University of Illinois at Urbana-Champaign
Department of Computer Science
Urbana, Illinois 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.

US NSF-GJ-328

2. Sponsoring Organization Name and Address

National Science Foundation
Washington, D. C. 20550

13. Type of Report & Period Covered

14.

5. Supplementary Notes

5. Abstracts

This report is concerned with the compilation of array expressions, such as appear in the language OL/2, Operator Language 2. The basic operation and operand types that may appear in an array expression in OL/2 are defined, as well as the data structures used to represent them. The parsing of source expressions into an intermediate expression tree form is described, and an algorithm for generating object code from the intermediate representation is given. The problem of reducing, through compilation techniques, the storage necessary for run time temporary results is discussed. The array expression compilation module for OL/2, as implemented, is presented in the Appendix.

'. Key Words and Document Analysis. 17a. Descriptors

Array Expression, Compilation,
Array Language,
Array Computation

b. Identifiers/Open-Ended Terms

c. COSATI Field/Group

. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages
nlimited	20. Security Class (This Page UNCLASSIFIED	22. Price